目錄

探索 **ES6**：升级至 **JavaScript** 的下一个版本

一本对 JavaScript 最新特性深入探讨的书籍，是 ECMAScript 2015 中新引入特性的一份全面指南。

为什么要翻译这本书？

很多人说，阮老师已经有一本关于 ES6 的书了 - 《ECMAScript 6 入门》，觉得看看这本书就足够了。

阮老师的那本书确实不错，值得 ES6 初学者去阅读。但是阮老师对这本书的定位是“中级难度”，也就是说书中不会很深入地去剖析各个知识点，而《Exploring ES6》这本书就努力在向大家细致地深入地讲解 ES6 的方方面面，这也是我觉得这本书很不错的原因。

此外，《Exploring ES6》的作者对 JavaScript 有比较深入全面的了解，也出了一本 ES5 及之前版本的书《Speaking JavaScript: An In-Depth Guide for Programmers》。

所以，总的来说，《Exploring ES6》还是值得一看的。

目录

点此处。

关于这本书

- 包含 ECMAScript 6（也被称为 ECMAScript 2015）可靠及深入的信息。
- 本书面向已经了解 JavaScript 的读者。

> 即使官方名称已经修改为 ECMAScript 2015，这本书也将保持 ECMAScript 6 的称呼——该称呼已广为人知。

如何加入翻译

1. fork 项目；
2. 提出 issue，表明自己想翻译哪些内容，时间大致是多久；
3. 组织者会审核此 issue，审核完成后会将 issue 设为 `翻译中` 状态，此时申请者可以开始翻译了；
4. 翻译完之后，提交 pull request，并将相应 issue 修改为 `翻译完成`；
5. pull request 审核通过后会被合并到 master 分支，issue 状态会被改为 `review 通过`；

6. 如果在指定时间内还没翻译完，则相应 issue 就会被设为 `超时未完成` 状态，此时该部分翻译内容可能会被重新分配给其他人。

Changelog

原书 Changelog

捐助

如果您觉得本翻译对您有帮助，并且愿意在金钱上给予翻译者鼓励，以间接的方式推进本书的翻译和校对，请扫一扫下面的支付宝二维码：



如果您觉得本翻译很垃圾，请在 issue 中提出来，或者直接提交 pull request ，对于您的批评和贡献，我们将不胜感激！

关于这本书你需要知道的

这本书是关于 ECMAScript 6（也被称为 ECMAScript 2015 ）的，它是 JavaScript 的新版本。

本书读者： **JavaScript** 程序员

为了读懂这本书，你应该已经知道 JavaScript 。如果不知道：我另外一本免费在线书《Speaking JavaScript》讲解了 JavaScript 的所有内容（直到并且包括 ECMAScript 5 ）。

为什么应该阅读这本书？

这本书包含了 ECMAScript 6 的详细内容，但是组织良好，以便于你能快速得到一个预览。它不仅讲解了 ES6 如何工作，也讲解了为什么要这么工作。

如何阅读这本书

这本书有很多内容，但是它的组织结构区分了不同难度级别的内容，很容易匹配上你的需求。有三种内容级别：

- 快速开始：绝大多数章以一个简明的概览开始，描述了本章节涉及到的特性。
- 坚实的基础：每一章都是以必需的内容开始，然后逐步进入细节。标题应该能让你明白什么时候停止阅读，但是我也偶尔在侧边栏给出小提示，告诉你了解某些内容的重要性。

1 关于 ECMAScript 6 （ ES6 ）

经过漫长的时间，最终，作为下一代 JavaScript 的 ECMAScript 6 总算变为现实：

- 在2015年6月成为标准。
- 浏览器的 JavaScript 引擎逐渐实现它的特性（参考 kangax 的 ES6 兼容表格）。
- 编译工具（比如 Babel 和 Traceur）能将 ES6 编译成 ES5 。

下一节讲解了在 ES6 世界中比较重要的几个概念。

1.1 TC39 （ Ecma 技术委员会 39 ）

TC39 （ Ecma 技术委员会 39 ） 是负责改进 JavaScript 的委员会。它的成员是公司（尤其是所有的主流浏览器厂商）。TC39 开会很频繁，参会者来自于成员公司的代表和邀请的一些专家。大会的部分现场视频可以在网上看到，这些视频展示了 TC39 是如何工作的。

1.2 ECMAScript 6 是如何设计的

ECMAScript 6 的设计过程集中在特性提案上。提案通常来自于开发者社区的建议。为了避免委员会参与设计，提案由志愿者（1 - 2名委员会的委托人）维护。

一个提案在成为标准之前会经历下面的步骤：

- 梗概（ Sketch ）（非正式地：“普通人提案”）：提案特性的第一个描述。
- 提案（ Proposal ）：如果 TC39 认为某个特性是重要的，那么此特性就上升为官方提案状态。这并不会保证最终会成为标准，但是大大地增加了成为标准的可能性。 ES6 提案的截止日期是2011年5月，在这之后不会考虑重大的新提案。
- 实现（ Implementations ）：提案特性必须被实现，在理想情况下，要支持两种 JavaScript 引擎。在提案获得提升的时候，来自于社区的实现和反馈决定了提案的样子。
- 标准（ Standard ）：如果提案持续检验自身，并且被 TC39 接受，那么该提案将最终包含进 ECMAScript 标准的一个版本中。此时，就成了一个标准特性。

[本节来源：《 The Harmony Process 》，作者 David Herman 。]

1.2.1 ES6 之后的设计进程

从 ECMAScript 7 （官方名字是 ECMAScript 2016 ）开始，TC39 将会按时发布每一版本。计划每年发布一个新的 ECMAScript 版本，此版本带有当时已就绪的特性。这意味着从现在开始， ECMAScript 版本将会是小的升级。

ECMAScript 2016 （以及接下来的版本）已经开始制定了，所有当前的提案都在 GitHub 上面列出。流程也已经改变了，在 TC39 流程文档中有描述。

1.3 JavaScript vs ECMAScript

大家都称呼这门语言是 JavaScript ，但是这个名字是一个商标（属于 Oracle 的，从 Sun 接手过来的）。因此， JavaScript 的官方名字是 ECMAScript 。这个名字来自于标准组织 Ecma ，该组织管理语言标准。既然 ECMAScript 这个名字获得大家的认可了，那么标准组织的名字就由缩写“ ECMA ”变为了“ Ecma ”。规范中定义的每一个 JavaScript 版本都采用该语言的官方名称。因此， JavaScript 标准的第一版本称为 ECMAScript 1 ，是“ ECMAScript Language Specification, Edition 1 ”的缩写。 ECMAScript x 也通常缩写为 ESx 。

1.4 升级到 ES6

web 的参与者包含：

- JavaScript 引擎的实现者
- web 应用的开发者
- 用户

这三种角色相互之间的约束控制很小。这就是为什么升级 web 语言很有挑战性。

一方面，升级引擎很有挑战，因为要应对运行于 web 的各种各样的代码，某些时候是很老的代码。你也希望引擎自动升级，尽量让用户感知不到。因此， ES6 是 ES5 的超集，没有移除任何东西。 ES6 升级了语言，但是并没有引入新的版本或者模式。它甚至使严格模式成为默认的执行模式（通过模块），而没有增加与非严格模式差异。采用的手法被称为“ One JavaScript ”，会在单独的一章里面讲解。

另一方面，升级代码具有挑战性，因为你的代码必须能在目标用户使用的各种各样的 JavaScript 引擎上面运行。因此，如果你想在你的代码中使用 ES6 ，仅有两个选择：等到目标用户不再使用非 ES6 的引擎，这将会等好几年；当 ES6 在2015年6月成为标准的事后，主流用户使用的引擎都支持 ES5 了（然而 ES5 早在2009年12月就成为标准了），因此可以将 ES6 代码编译成 ES5 代码，现在就用起来（关于怎么做，会在单独的一章中讲解）。

目标和需求在 ES6 的设计中产生了冲突：

- 目标是修复 JavaScript 的陷阱，并且添加新的特性。
- 需求是既要让老代码能够运行，又要保持语言的轻量性。

1.5 ES6 的目标

Harmony/ES6 的原始项目有几个目标。在下面的子节里，我会讲解其中一些目标。

1.5.1 目标：成为一门更好的语言

目标是：成为一门对编写如下程序更友好的语言：

- 1、复杂的应用；
- 2、应用之间共享的库（可能包含 DOM 操作）；
- 3、针对新版本的代码生成器。

子目标（1）承认了用 JavaScript 开发的应用已经变得很大了。达成这个目标的关键 ES6 特性就是内置模块。

模块也是目标（2）的解决方案。说句题外话，在 JavaScript 里实现 DOM 是出了名的困难。 ES6 的 Proxy 应该会有所帮助（这会在单独的一章中讲解）。

特意添加了几个特性，这些特性不是为了提升 JavaScript ，而是使其它语言更容易编译成 JavaScript 。举两个例子：

- Math.fround() - 舍入为32位的浮点数
- Math.imul() - 两个32位整数相乘

它们对于将其它语言（例如 C/C++ 通过 Emscripten 编译）编译成 JavaScript 是很有用的。

1.5.2 目标：提升互操作性

目标是：提升互操作性，尽可能采用事实上的标准。

四个例子：

- 类：基于现今构造器的使用方式。
- 模块：从 CommonJS 模块格式中提取设计思想。
- 箭头函数：从 CoffeeScript 中借用过来的语法。
- 命名函数参数：对于命名参数，没有内置的支持。但是，可以通过对象字面量结合参数定义的解构来实现命名参数。

1.5.3 目标：版本控制

目标是：尽量使版本控制简单并且线性的。

正如前面提到的， ES6 避免“ One JavaScript ”式的版本管理：在一个 ES6 的代码库里，所有代码都是 ES6 的，没有任何一部分是 ES6 特有的。

1.6 ES6 特性概览

引用 ECMAScript 6 规范中的一段介绍：

> ECMAScript 6 的一些主要增强功能，包括模块、类声明、词块作用域、迭代器、生成器、用于异步编程的 Promise 、解构模式和应该有的尾递归调用。ECMAScript 内置库已经开始支持其它的抽象数据，包括 Map ， Set ， 二进制数字值数组，对 Unicode 字符串中字符的额外支持，正则表达式。内置类型现在可以通过继承来扩展了。

有三组主要特性：

- 对于已经存在的特性（例如通过库存在的特性）提供更好的语法。例如：
  - 类
  - 模块
- 标准库新的功能。例如：
  - 字符串和数组的新方法
  - Promise
  - Map ， Set
- 全新的特性。例如：
  - 生成器
  - Proxy
  - WeakMap

1.7 ECMAScript 简史

本节叙述了发展到 ECMAScript 6 的过程中所发生的事情。

1.7.1 早些年：ECMAScript 1–3

- ECMAScript 1 （1997年6月）是 JavaScript 语言规范的第一个版本。
- ECMAScript 2 （1998年6月）包含了一些小的变化，为了使规范与独立的 JavaScript ISO 标准保持同步。
- ECMAScript 3 （1999年12月）引入了很多特性，已经成为语言的流行部分。

1.7.2 ECMAScript 4 （在2008年6月被抛弃）

在1999年 ES3 发布之后，就开始了 ES4 的制定工作。在2003年，发布了一份临时报告，在这之后 ES4 的制定工作就暂停了。 Adobe 的 ActionScript 和微软的 JScript.NET 实现了临时报告中所描述语言的子级。

在2005年2月， Jesse James Garrett 发现新技术变得流行起来，被用于实现动态的使用 JavaScript 的前端 app 。他称呼这些技术为 Ajax 。 Ajax 开启了 web app 的全新世界，并使人们对 JavaScript 产生了浓厚的兴趣。

这使得 TC39 在2005年秋天重启 ES4 的制定工作。他们基于 ES3 来制定 ES4 ，临时的 ES4 在 ActionScript 和 JScript.NET 中实现。

当时有两个组从事制定未来 ECMAScript 版本的工作：

- ECMAScript 4 由 Adobe ， Mozilla ， Opera 和 Google 设计，是一个很大的升级。它包含的特性集如下：
  - 大型编程（类，接口，命名空间，包，程序单元，可选的类型注释，可选的静态类型检测和验证）
  - 编程和脚本的进化（结构类型，鸭子类型，类型定义，和方法重载）
  - 数据结构完善（参数化类型， getter 和 setter ，和元级的方法）
  - 控制抽象（恰当的尾递归，迭代器，和生成器）
  - 反思（类型元对象和堆栈标记）
- ECMAScript 3.1 由微软和 Yahoo 设计。计划成为 ES4 的子集，并且是 ECMAScript 3 的增量化升级，没有 bug 修复和小的新特性。 ECMAScript 3.1 最终成为了 ECMAScript 5 。

这两个组关于 JavaScript 未来产生了分歧，他们之间的紧张局势也逐渐升级。

> 本节来源：
>
> 《 Proposed ECMAScript 4th Edition – Language Overview 》，2007-10-23
> 《 ECMAScript Harmony 》，作者 John Resig ，2008-08-13

1.7.3 ECMAScript Harmony

2008年7月底，TC39 在 Oslo 开了个会， Brendan Eich 描述会议结果如下：

> JavaScript 标准正文不是秘密， Ecma 的技术委员会39已经分裂超过一年了，一些成员喜欢 ES4 [...] ，一些成员倡导 ES3.1 [...] 。现在，我很高兴地说，分裂结束了。

会议就四点上达成了一致：

- 1、开发一个增量升级的 ECMAScript （就是 ECMAScript 5 ）。
- 2、开发一个主要的新版本，相对于 ES4 更加现代化，但是在范围上比 ECMAScript 3 之后的版本大得多。这个版本代号为 *Harmony* ，名字源于会议要达成的消除分歧的目标。
- 3、来自于 ECMAScript 4 的特性被取消了一部分：包，命名空间，早绑定。
- 4、其他想法在 TC39 达成一致后才开发。

因此： ES4 小组同意不让 Harmony 像 ES4 那么激进，剩下的 TC39 成员赞成继续推进。

接下来的 ECMAScript 版本是：

- ECMAScript 5 （2009年12月）。这是目前大多数浏览器实现的 ECMAScript 版本。它给标准库引入了几处增强，并且通过 严格模式 升级了语言的语义。
- ECMAScript 6 （2015年6月）。该版本经历了几次名字修改：
  - ECMAScript Harmony ： 是最初的代号，用于 ECMAScript 5 之后的 JavaScript 的改进。
  - ECMAScript.next ： 很明显， Harmony 的计划对于一次版本升级来说显得太雄心勃勃了，因此它的特性分成了两组：第一组有很高的优先级，成为继 ES5 之后的版本。为了避免过早地涉及到版本号，这个版本的代号就是 ECMAScript.next ， ES4 就因为这个产生了很多问题。第二组特性直到 ECMAScript.next 之后才有时间了。
  - ECMAScript 6 ： 当 ECMAScript.next 成熟之后，它的代号就弃用了，大家都开始称其为 ECMAScript 6 。
  - ECMAScript 2015 ： 在2014年底， TC39 决定将官方的 ECMAScript 6 名字修改为 ECMAScript 2015 ，根据接下来的每年发布规范的策略。
- ECMAScript 2016 之前被叫做 ECMAScript 7 。开始于 ES2016 ，语言标准将会每年发布一个小的版本。

2 ECMAScript 6 常见问题解答

本章解答了几个常问的关于 ECMAScript 6 的问题。

2.1 当前引擎支持 ES6 情况如何？

检查 ES6 在各引擎上的支持情况的最好方式就是去查看 Kangax 的 ES6 兼容表格。

2.2 如何将 ECMAScript 5 代码升级至 ECMAScript 6 ？

什么都不用做： ECMAScript 6 是 ECMAScript 5 的超集。因此，所有的 ES5 代码自动的就是 ES6 代码了。关于 ES6 是如何保持向后兼容性的，在关于“ One JavaScript ”的那一章讲解。

如果你想知道现在如何使用新的 ES6 特性（这是另一个的问题），转到章节“部署 ECMAScript 6 ”。

2.3 现在学习 ECMAScript 5 还有意义吗？

正如“部署 ECMAScript 6 ”那一章讲述的，你现在就可以使用 ES6 编写代码了，无需在旧的 JavaScript 版本下编写代码。难道这意味着不再需要学习 ECMAScript 5 了吗？不是这样的，有这样几个理由：

- ECMAScript 6 是 ECMAScript 5 的超集 - 新的 JavaScript 版本不能破坏已有的代码。因此，学习 ECMAScript 5 不是徒劳的。
- 有几个 ECMAScript 6 特性特换掉了 ECMAScript 5 的特性，但是仍然可以使用它们作为基本原理。理解这些原理很重要。两个例子：类在内部转换成构造器，并且方法依然是函数（就像以前一样）。
- 只要 ECMAScript 6 编译成了 ECMAScript 5 ，对于理解编译过程很有用。并且你可能在未来好几年都要像这样编译成 ES5 ，直到 ES6 在相应浏览器中得到了支持。
- 能够看懂遗留的代码很重要。

2.4 ES6 臃肿吗？

我看见人们抱怨 ES6 使 JavaScript 变得臃肿，包含了太多无用的“语法糖”（为已经存在的东西制定的更方便的语法）。

如果有人觉得这样的话，我建议使用 ES6 一段时间。没人强迫你使用任何新的特性。你可以从一些小的特性开始使用（例如模板字面量和箭头函数），然后在你对 ES6 感到更舒服的时候，使用更多的新特性。到目前为止，我从真正使用了 ES6 的人（不是仅仅在书中阅读过的人）那里得到的反馈来看，大多数还是很积极的。

此外，表面上看起来是语法糖的特性（比如类和模块），引入了很多需要的标准到语言中，并作为未来特性的基础。

最后，有几个特性并不是为普通开发者创建的，是为库的开发者创建的（例如生成器，迭代器， Proxy 等）。“普通开发者”只需要粗略地知道就行了。

2.5 ES6 规范文档不是很长吗？

ECMAScript 规范文档的确变长了很多： ECMAScript 5.1 PDF 有245页， ES6 PDF 有593页。但是，作为对比， Java 8 语言规范有724页（不包括索引）。而且， ES6 规范包含了实现细节，而其他语言省略了。规范也说明了标准库是如何工作的。

2.6 ES6 包含了数组生成表达式（ Array Comprehension ）吗？

本来， ES6 计划纳入数组和生成器的生成表达式（类似于 Haskell 和 Python ）。但是被推迟到 ES6 之后，因为 TC39 想思考两个东西：

- 可能需要能够创建生成表达式，对任何数据类型都有效（参考微软的 LINQ ）。
- 给迭代器添加相应的方法来实现生成表达式所实现的功能可能更好。

2.7 ES6 是静态类型的吗？

静态类型不是 ES6 的一部分。然而，下面的两种技术给 JavaScript 引入了静态类型的功能。类似的特性可能最终会被标准化。

- 微软的 TypeScript ：基本上是 ES6 加上可选的类型注解。此时，如果这样做的话，编译成的 ES5 会丢掉类型信息。也可以使类型信息在运行时可用，这些信息用于类型反查和运行时类型检测。
- Facebook Flow ：是一个 ECMAScript 6 的类型检测器，基于流分析（ flow analysis ）。因此，它仅给语言增加了可选的类型注解，以及推理类型和检查类型，并不会帮助将 ES6 代码编译成 ES5 代码。

静态类型的三个好处是：

- 能够提前检测到某些错误，因为代码可以被静态分析（在开发阶段，不用执行代码）。因此，静态类型和测试、捕获不同类型错误是互补的。
- 能够帮助 IDE 自动补全。

TypeScript 和 Flow 使用相同的注解。类型注解是可选的，这使得这种方式相当轻量。即便没有注解，通常也能够推断类型。因此，这种类型检查甚至对于完全没注解的代码也很有用，用作一种辅助的检查。

2.8 应该避免使用类吗？

我推荐使用类。我在关于类的那一章讲解了它的优点和缺点。

2.9 ES6 有 `特性（ traits ）` 或者 `混入` （ mixins ）吗？

没有，但是目前类的一个目标就是作为将来可能的 `特性（ traits ）` （或者类似的可以做多继承的方式）的一个基础。

可以从库 traits 中初步看到 `特性（ traits ）` 的样子。这是一个库，因此限制了语法格式；只要 `特性（ traits ）` 成为语言的特性，就会有更加漂亮的语法了。

2.10 为什么 ES6 有带 `=>` 的箭头函数，而没有带 `->` 的箭头函数？

ECMAScript 6 对于函数有词法 `this` 的语法，就是所谓的箭头函数。但是，对于函数没有动态 `this` 的箭头函数语法。这是故意的；方法定义覆盖了带 `->` 箭头的函数的绝大多数使用场景。如果真的需要动态的 `this` ，你可以使用传统的函数表达式。

2.11 在哪里可以找到更多的 ES6 资源？

有两组 ES6 资源：

- “ ECMAScript 6 Tools ”，作者 Addy Osmani 。
- “ ECMAScript 6 Learning! ”，作者 Eric Douglas 。

3 一个 JavaScript ： 在 ECMAScript 6 中避免版本化

给一门语言添加新特性的最好方式是什么？本章描述了 ECMAScript 6 使用的方式。它被称作一个 *JavaScript* ，因为避免了版本化。

3.1 版本化

原则上，语言的一个新版本是一个清理的机会，可以清理过时的特性或者改变特性的工作方式。这意味着新的代码在语言的旧的实现中无法工作，老的代码在新的实现中无法工作。每段代码都和特定的语言版本关联。针对两个不同语言版本写两种不同代码是很常见的。

首先，你可以使用一种“用所有还是什么都不用”的方式：如果一个代码库需要使用新的语言版本，就必须彻底升级。Python 在从 Python 2 升级到 Python 3 的时候就是这样的。这样的话有个问题，将已有的代码库一次性全部升级可能是做不到的，尤其是代码库很大的时候。而且，这种方式对于 web 来说是不可行的，因为总是有老代码，而且 JavaScript 引擎会自动升级。

第二，你可以让一个代码库包含在多个语言版本都可运行的代码，通过根据版本切换代码的方式。你可以通过一个专用的互联网媒体类型标记 ECMAScript 6 的代码。这样的媒体类型可以和一个 HTTP 头关联在一起：

```
Content-Type: application/ecmascript;version=6
```

也可以和 `<script>` 元素中的 `type` 属性关联在一起（ `type` 默认值是 `text/javascript` ）：

```
<script type="application/ecmascript;version=6">
    ...
</script>
```

这是在代码之外指定版本。另一种方案是在代码内指定版本。例如，将下面的代码放在 JavaScript 文件的第一行：

```
use version 6;
```

两种标记的方法都很容易产生问题：外部版本标记法很脆弱，容易丢失；内部版本标记法又会使代码显得杂乱。

一个更根本的问题是，针对不同的语言版本，要维护不同的执行引擎。这就引起了一些问题：

- 引擎变得臃肿，因为要实现所有版本的语义。对于分析语言的工具也带来了同样的问题（比如类型检测， JSLint ）。
- 开发者需要记住版本之间的不同点。
- 代码变得更加难以重构，因为在移动代码的时候需要考虑语言版本的问题。

因此，应该避免版本化，尤其是 JavaScript 和 web 。

3.1.1 非版本化的升级

但是我们如何解决版本化的问题？通过一直向后兼容。这意味着我们必须放弃一些野心，比如清理 JavaScript ：我们不能引入破坏性改变。向后兼容意味着不要移除特性，也不要修改特性。原则是："不要破坏 web "。

然而，我们可以添加新的特性，并且使已有的特性更加强大。

因此，对于新的引擎不需要版本化管理了，因为仍然可以运行老的代码。 David Herman 称这种避免版本化的方法为 `一个 JavaScript （ One JavaScript(1JS) [1] ）` ，因为它使 JavaScript 避免分成几种不同的版本或模式。正如我们将要看到的，由于严格模式，使得 1JS 甚至移除了一些已有的分裂。

一个 JavaScript （ One JavaScript ）不是说你必须完全放弃语言清理。你可以引入新的干净的特性，而不是清理掉已有的特性。其中一个例子就是 `let` ，它声明了块级范围的变量，是 `var` 的一个升级版本。它并不取代 `var` ，而是作为高级可选项一直和 `var` 并存。

某一天，甚至可能清除掉没人使用的特性。一些 ES6 特性是在调查过 web 上的 JavaScript 代码才设计的。举两个例子：

- `let` 声明很难添加到非严格模式，因为 `let` 在这种模式下是保留字。使用 `let` 关键字看起来像是合法 ES5 代码的形式是：

  ```
  let[x] = arr;
  ```

  经研究得出，在 web 上没人以这种方式在非严格模式下使用变量 `let` 。这使得 TC39 将 `let` 添加进了严格模式。在本章后面讲述了这是如何做的。

- 函数声明确实偶尔会在非严格模式下的代码块中出现，这就是为什么 ES6 规范描述了 web 浏览器可以采用的措施来确保这样的代码不会被破坏。后面详细讲解。

下一章描述了我们刚才看到的反面：部署 ES6 ，以便能够在支持 ES6的引擎上运行，也能够在不支持的引擎上面运行。

3.2 严格模式与 ECMAScript 6

ECMAScript 5 引入严格模式来清理语言，在文件或者函数的第一行放入下面的内容就可以开启严格模式：

```
'use strict';
```

严格模式引入了三种破坏性的改变：

- 语法改变：一些之前合法的语法在严格模式下面是不允许的。例如：
    - 禁止 `with` 语句。它允许使用者添加任何对象到变量作用域链，这会减缓程序的执行速度，并且很难指出某个变量指向哪里。
    - 删除一个独立的标识符（一个变量，而不是一个属性）是不允许的。
    - 函数只能在作用域的顶层声明。
    - 更多的保留字： implements interface let package private protected public static yield 。
- 更多种类的错误。例如：
    - 给一个未声明的变量赋值会抛出 `ReferenceError` 。在非严格模式下，这样干就会创建一个全局变量。
    - 修改只读的属性（比如字符串的长度属性）会抛出 `TypeError` 。在非严格模式下，不会产生任何效果。
- 不同的语义：在严格模式下，一些语法结构表现得不一样。例如：
    - `arguments` 不再随着当前参数值的改变而改变。
    - 在非方法的函数中 `this` 是 `undefined` 。在非严格模式下，它指向全局对象（ `window` ），也就是说如果调用一个构造器的时候没有使用 `new` ，就会创建一些全局变量。

严格模式是一个很好地说明了版本化是棘手的：即便能够制作一个干净版本的 JavaScript ，也很难被大家接受。主要原因是破坏了一些现存的代码，降低了执行速度，并且加入到文件中也很麻烦（更不用说交互的命令行）。我喜欢严格模式这种想法，但是基本不使用。

3.2.1 支持松散（非严格）模式

一个 JavaScript 意味着我们不能放弃松散模式：此模式将会继续存在（例如在 HTML 属性中）。因此，我们不能基于严格模式来构建 ECMAScript 6 ，必须同时在严格模式和非严格模式（又称为松散模式）下添加特性。否则，严格模式就会成为一个不同的语言版本，就退回了版本化的方式。很不幸，有两个特性很难引入松散模式： `let` 声明和块级函数声明。让我们看看为什么很难引入和如何引入。

3.2.2 松散模式中的 `let` 声明

`let` 使你能够声明块级变量。这很难被引入到松散模式，因为 `let` 仅在严格模式下是保留字。也就是说，下面两条语句在 ES5 的松散模式下是合法的：

```
var let = [];
let[x] = 'abc';
```

在 ECMASCript 6 的严格模式下，第一行就会抛出异常。因为使用了 `let` 作为变量名。然后第二行会被解析为一个 `let` 变量声明（使用解构）。

在 ECMAScript 6 的松散模式下，第一行不会抛出异常，但是第二行依然被解析为一个 `let` 声明。这种使用 `let` 的方式在 web 上是极少见的，因此 ES6 可以直接这样来解析。 ES5 松散模式下的其他 `let` 声明的书写方式不会被误解：

```
let foo = 123;
let {x,y} = computeCoordinates();
```

3.2.3 松散模式下的块级函数声明

ECMAScript 5 严格模式禁止在块中声明函数，在松散模式下，规范却允许这么做，但是没说这样会发生什么。因此，很多 JavaScript 实现都支持块级函数声明，但是处理方式是不一样的。

ECMAScript 6 想要块中的函数声明本地化（即该函数的作用域救在该块中）。作为 ES5 严格模式的扩展，这是没问题的，但是破坏了一些松散模式的代码。因此， ES6 为浏览器提供了“web 遗留的兼容语义”，允许块中的函数声明在函数作用域中存在。

3.2.4 其它关键字

标识符 `yield` 和 `static` 仅在 ES5 的严格模式下是保留字。 ECMAScript 6 使用上下文相关的语法规则来使它们在松散模式下起作用：

- 在松散模式下， `yield` 仅在生成器函数中是保留字。
- `static` 现在仅用于类字面量中，类字面中默认就是严格的（见下文）。

3.2.5 隐式的严格模式

在 ECMAScript 6 中，模块体和类体默认就是严格模式的–没必要使用 `use strict` 标记。考虑到我们所有的代码都将会位于模块中， ECMAScript 6 有效地将整个语言升级到了严格模式。

其它结构体（比如箭头函数和生成器函数）本来也应该隐式地为严格模式。但是考虑到通常情况下这些结构都很小，在非严格模式下使用它们就会造成代码中两种模式的碎片化切换。类，尤其是模块一般是足够大的，这样一来就可以忽略碎片化的代码片段问题了。

3.2.6 无法修复的东西

一个 JavaScript 的缺陷就是无法修复已有的怪异行为，尤其是下面这两个。

第一个，`typeof null` 应该返回字符串 `null` 而不是 `object` 。但是修正这个就会破坏已有的代码。另一方面，给新种类的操作数添加新的操作结果是没问题的，因为当前的 JavaScript 引擎对于一些宿主对象已经会返回自定义的值。ECMAScript 6 的 Symbol 就是一个例子：

```
> typeof Symbol.iterator
'symbol'
```

第二个，全局对象（浏览器中的 `window` 对象）不应该在变量作用域链。但是现在修正这个也太晚了。但是至少，在模块中不会处于全局作用域下，并且 `let` 永远不会创建全局对象属性，甚至在全局作用域下使用也不会。

3.3 总结

一个 JavaScript 意思就是使 ECMAScript 6 完全地向后兼容，很高兴这获得了成功。尤其是模块隐式就是严格模式的（这样一来我们大部分的代码都会处于严格模式下）。

在短期内，对于制定 ES6 规范和引擎实现来说，给严格模式和松散模式添加 ES6 的语法结构会耗费更多的精力。从长远来看，规范和引擎将会受益于语言不分叉（更少的膨胀等等）。开发人员会立即从一个 JavaScript 中获得好处，因为开始使用 ECMAScript 6 变得更加容易。

3.4 深入阅读

〔1〕 原始的 1 JS 提案（警告：已过时）：“ ES6 doesn’t need opt-in ”，作者 David Herman 。

4 进入 ECMAScript 6 的第一步

本章帮助你进入 ECMAScript 6 的开发：

- 讲解了如何交互式地尝试 ES6 。
- 列出了易于理解的 ES6 特性，顺便讲解了这些特性在 ES5 中如何实现。

4.1 尝试 ECMAScript 6

有三种简单的方式来玩儿 ES6 ：

- 1、 web 浏览器：使用在线的 Babel REPL ，一个交互式的工具，能将 ES6 编译成 ES5 。如果选择这种方式的话就什么都不用安装。
- 2、命令行：使用 `babel-node` ，一个 Node.js 的可执行版本，它能运行 ES6 代码（在内部编译成 ES5 ）。它能通过 npm 安装。
- 3、各个 JavaScript 引擎：参考 kangax 的 ES6 兼容表格，这个表格能查出在某个引擎中本地支持哪些 ES6 特性。

后续将会更详细地介绍第一和第二选项。

5 新的数值和 Math 特性

本章描述了 ECMAScript 6 的新的数值和 Math 特性。

5.1 概览

现在可以写二进制和八进制数值字面量：

```
> 0xFF // ES5: hexadecimal
255
> 0b11 // ES6: binary
3
> 0o10 // ES6: octal
8
```

全局对象 Number 上添加了一些新的属性，尤其是：

- `Number.EPSILON` 用于比较浮点数，容忍舍入错误。
- 一个方法和常量，用于判断 JavaScript 整数是否安全（在有符号的53位范围内，没有精度的损失）。

5.2 新的整型字面量

ECMAScript 5 已经有了十六进制的整型字面量：

```
> 0x9
9
> 0xA
10
> 0x10
16
> 0xFF
255
```

ECMAScript 6 引入了两种新的整型字面量：

- 以 `0b` 或 `0B` 开头的二进制字面量：

```
> 0b11
3
> 0b100
4
```

- 以 `0o` 或 `0O` 开头的八进制字面量（是的，就是一个零后加大写字母 O；使用第一种小写字母 o 的形式也是可以的）：

```
> 0o7
7
> 0o10
8
```

记住方法 Number.prototype.toString(radix) 可用于将数值转换回来：

```
> (255).toString(16)
'ff'
> (4).toString(2)
'100'
> (8).toString(8)
'10'
```

5.2.1 八进制字面量的使用场景：Unix 风格的文件权限

在 Node.js 的文件系统模块，有几个函数有 `mode` 参数，用于指定文件权限，via an encoding that is a holdover from Unix：

- 权限被指定给三种用户：
  - 用户：文件拥有者
  - 组：跟文件相关的组成员
  - 所有：其余用户
- 每一种用户，都可以被授予下列权限：
  - r（read）：此种用户允许读文件
  - w（write）：此种用户允许改变文件
  - x（execute）：此种用户允许运行文件

这意味着权限可以通过9位来表示（3种用户，每种用户有三种权限）：

	用户	组	所有
权限	r，w，x	r，w，x	r，w，x
位（Bit）	8，7，6	5，4，3	2，1，0

某一种类型的用户的权限值占3位：

位	权限	八进制数字
000	---	0
001	--x	1
010	-w-	2
011	-wx	3
100	r--	4
101	r-x	5
110	rw-	6
111	rwx	7

这表明八进制数很适合用来代表所有权限，仅需要3个数字，每个数字代表一种用户的权限。两个例子：

- 755 = 111,101,101：我可以修改，读取和执行；其他所有人只能读取和执行。
- 640 = 110,100,000：我可以读取和写入；组成员可以读取；其余用户根本不能获取。

5.2.2 `parseInt()` 和新的整型字面量

`parseInt()` 有如下声明：

```
parseInt(string, radix?)
```

它对十六进制字面量提供了特殊的支持 - 当满足下述条件的时候，`string` 参数的 `0x` （或 `0X` ）前缀会被移除：

- 未传递 `radix` 参数或者该参数为0，此时 `radix` 被设为16。
- 传递 `radix` 并为16。

例如：

```
> parseInt('0xFF')
255
> parseInt('0xFF', 0)
255
> parseInt('0xFF', 16)
255
```

在所有其他情形下，数字一直解析到第一个非数字的地方：

```
> parseInt('0xFF', 10)
0
> parseInt('0xFF', 17)
0
```

`parseInt()` 并没有对八进制或二进制字面量特殊支持！

```
> parseInt('0b111')
0
> parseInt('0b111', 2)
0
> parseInt('111', 2)
7

> parseInt('0o10')
0
> parseInt('0o10', 8)
0
> parseInt('10', 8)
8
```

如果想解析这种字面量，应该使用 `Number()` ：

```
> Number('0b111')
7
> Number('0o10')
8
```

或者，也可以移除前缀，然后使用 `parseInt()` ，传入适当的 `radix` 参数：

```
> parseInt('111', 2)
7
> parseInt('10', 8)
8
```

5.3 新的静态 Number 类属性

本节叙述了 ECMAScript 6 中添加的构造器 Number 上的新属性。

5.3.1 预定义的全局函数

四个数值相关的全局函数已经添加到 `Number` 下面了，这些方法是：`isFinite` 和 `isNaN` ，`parseFloat` 和 `parseInt` 。所有方法都表现得基本跟对应的全局函数一样，但是 `isFinite` 和 `isNaN` 不需要它们的参数是数字。下述子章节解释了所有细节。

5.3.1.1 `Number.isFinite(number)`

`number` 是否是一个真正的数字（不为 `Infinity` 或 `-Infinity` 或 `NaN` ）？

```
> Number.isFinite(Infinity)
false
> Number.isFinite(-Infinity)
false
> Number.isFinite(NaN)
false
> Number.isFinite(123)
true
```

该方法的优点是它并不需要它的参数是数字（全局的那个函数需要）：

```
> Number.isFinite('123')
false
> isFinite('123')
true
```

5.3.1.2 `Number.isNaN(number)`

`number` 是否为 NaN ？通过 `===` 来检查是奇巧淫技。 NaN 是唯一一个自己不等于自己的值：

```
> let x = NaN;
> x === NaN
false
```

因此，下述表达式用于检查是否为 `NaN` ：

```
> x !== x
true
```

使用 `Number.isNaN()` 具有更强的自我描述力：

```
> Number.isNaN(x)
true
```

`Number.isNaN()` 也不需要它的参数是数字（全局的那个函数需要）：

```
> Number.isNaN('???')
false
> isNaN('???')
true
```

5.3.1.3 `Number.parseFloat` 和 `Number.parseInt`

下面两个方法和全局的同名函数效果一样。由于完整性的元婴，它们被添加到 `Number` 上；现在，所有数字相关的函数都在 `Number` 上了。

- `Number.parseFloat(string)`
- `Number.parseInt(string, radix)`

5.3.2 Number.EPSILON

舍入误差在 JavaScript 中是一个问题，尤其是十进制小数。例如，0.1和0.2可以被精确地描述，如果两者相加，并和0.3比较，会有如下结果：

```
> 0.1 + 0.2 === 0.3
false
```

在比较浮点数的时候，`Number.EPSILON` 指定了一个合理的误差范围。它提供了一种更好的方式来比较浮点数值，就像如下函数所示：

```
function epsEqu(x, y) {
    return Math.abs(x - y) < Number.EPSILON;
}
console.log(epsEqu(0.1+0.2, 0.3)); // true
```

5.3.3 `Number.isInteger(number)`

JavaScript 仅有浮点数（双精度）。相应的，整数是简单的没有小数部分的浮点数。

如果 `number` 是数字并且没有小数部分，则 `Number.isInteger(number)` 返回 `true` 。

```
> Number.isInteger(-17)
true
> Number.isInteger(33)
true
> Number.isInteger(33.1)
false
> Number.isInteger('33')
false
> Number.isInteger(NaN)
false
> Number.isInteger(Infinity)
false
```

5.3.4 安全整数

JavaScript 数字仅有足够的空间存放53位有符号整数。也就是说，在范围$-2^{53} < i < 2^{53}$之间的整数 i 是安全的。先暂时解释这究竟意味着什么。下面的属性帮助判断某个 JavaScript 整数是否是安全的：

- `Number.isSafeInterger(number)`
- `Number.MIN_SAFE_INTEGER`
- `Number.MAX_SAFE_INTEGER`

安全整数的概念重点在于数学上的整数在 JavaScript 中如何展示。在范围（-2^{53}，2^{53}）（除去上下边界）中，JavaScript 整数是安全的：要展示的数学上的整数和范围中的整数是一一对应的。

超过这个范围，JavaScript 整数就不安全了：两个或者多个数学上的整数使用同一个 JavaScript 整数来表示。例如，从2^{53}开始，JavaScript 仅能表示数学上的偶数：

```
> Math.pow(2, 53)
9007199254740992

> 9007199254740992
9007199254740992
> 9007199254740993
9007199254740992
> 9007199254740994
9007199254740994
> 9007199254740995
9007199254740996
> 9007199254740996
9007199254740996
> 9007199254740997
9007199254740996
```

因此，安全整数能够明确代表某一个数学上的整数。

5.3.4.1 `Number` 上跟安全整数相关的静态属性

这两个静态的 `Number` 属性指定了安全整数的上下边界，可能像如下所示的方式定义：

```
Number.MAX_SAFE_INTEGER = Math.pow(2, 53)-1;
Number.MIN_SAFE_INTEGER = -Number.MAX_SAFE_INTEGER;
```

`Number.isSafeInteger()` 用于判定一个 JavaScript 数字是否是安全整数，可以这样来定义该方法：

```
Number.isSafeInteger = function (n) {
    return (typeof n === 'number' &&
        Math.round(n) === n &&
        Number.MIN_SAFE_INTEGER <= n &&
        n <= Number.MAX_SAFE_INTEGER);
}
```

对于给定的值 n，该函数首先检查 `n` 是否是数字和整数。如果两次检查都通过了，并且 `n` 大于等于 `MIN_SAFE_INTEGER`，小于等于 `MAX_SAFE_INTEGER`，那么 `n` 就是安全的。

5.3.4.2 什么时候整数运算是正确的？

如何确定整数运算的结果是正确的？例如，下面的结果明显不对：

```
> 9007199254740990 + 3
9007199254740992
```

两个操作数都是安全的，但是得到了一个不安全的结果：

```
> Number.isSafeInteger(9007199254740990)
true
> Number.isSafeInteger(3)
true
> Number.isSafeInteger(9007199254740992)
false
```

下面的结果依然不正确：

```
> 9007199254740995 - 10
9007199254740986
```

这次，结果是安全的，但是其中一个操作数不安全：

```
> Number.isSafeInteger(9007199254740995)
false
> Number.isSafeInteger(10)
true
> Number.isSafeInteger(9007199254740986)
true
```

因此，对整数应用操作符 `op` ，仅当所有操作数和结果是安全的，才能确保结果是正确的。更正式地：

```
isSafeInteger(a) && isSafeInteger(b) && isSafeInteger(a op b)
```

意味着 `a op b` 能得到正确的结果。

> 本节来源 "Clarify integer and safe integer resolution"， Mark S. Miller 发送给 es 讨论邮件列表。

5.4 Math

ECMAScript 6 中的全局对象 Math 有一个新的方法。

5.4.1 各种各样的数值函数

5.4.1.1 `Math.sign(x)`

返回 `x` 的符号，-1或+1。如果 `x` 是 `NaN` 或者零，则返回 x 。

```
> Math.sign(-8)
-1
> Math.sign(3)
1

> Math.sign(0)
0
> Math.sign(NaN)
NaN

> Math.sign(-Infinity)
-1
> Math.sign(Infinity)
1
```

6 新的字符串特性

本章覆盖了 ECMAScript 6 中字符串所有的新特性。

6.1 概览

新的字符串方法：

```
> 'hello'.startsWith('hell')
true
> 'hello'.endsWith('ello')
true
> 'hello'.includes('ell')
true
> 'doo '.repeat(3)
'doo doo doo '
```

ES6 有一种新的字符串字面量，模板字面量：

```
// String interpolation via template literals (in backticks)
let first = 'Jane';
let last = 'Doe';
console.log(`Hello ${first} ${last}!`);
    // Hello Jane Doe!

// Template literals also let you create strings with multiple l
ines
let multiLine = `
This is
a string
with multiple
lines`;
```

6.2 Unicode 码点解析

在 ECMAScript 6 中，存在一种新的 Unicode 码点解析方式：能够指定任何编码方式（即便这种编码方式超过了16位）：

```
console.log('\u{1F680}');    // ES6: 一个码点
console.log('\uD83D\uDE80'); // ES5: 两个码点单元
```

本章关于 Unicode 的部分展示了更多信息。

6.3 字符串插值，多行字符串字面量和原始字符串字面量

模板字面量有专门的章节深入讲述，其提供了三种有意思的特性。

首先，模板字面量支持字符串插值：

```
const first = 'Jane';
const last = 'Doe';
console.log(`Hello ${first} ${last}!`);
    // Hello Jane Doe!
```

其次，模板字面量可以跨多行：

```
const multiLine = `
This is
a string
with multiple
lines`;
```

最后，如果在模板字面量前面加上标签 `String.raw` ，那么就变成了“原生的”的字符串了，反斜杠再也不是用于转义的特殊字符，也就是说， `\n` 不会转义成换行符：

```
const str = String.raw`Not a newline: \n`;
console.log(str === 'Not a newline: \\n'); // true
```

6.4 字符串遍历

字符串是可遍历的，也就是说可以使用 `for-of` 遍历字符串中的字符：

```
for (const ch of 'abc') {
    console.log(ch);
}
// Output:
// a
// b
// c
```

也可以使用扩展操作符（...）将字符串转换成数组：

```
const chars = [...'abc'];
    // ['a', 'b', 'c']
```

6.4.1 字符串遍历遵循 Unicode 码点规则

字符串遍历的时候，分割字符串是按照码点边界来的，这意味着遍历的时候拿到的字符可能由一个或者两个 JavaScript 字符组成：

```
for (const ch of 'x\uD83D\uDE80y') {
    console.log(ch.length);
}
// Output:
// 1
// 2
// 1
```

6.4.2 计算码点数

遍历是一种快速计算字符串 Unicode 码点数的方式：

```
> [...'x\uD83D\uDE80y'].length
3
```

6.4.3 翻转包含非 BMP 码点的字符串

利用遍历也能够实现翻转包含非 BMP 码点（比16位大，用两个 JavaScript 字符编码）的字符串：

```
const str = 'x\uD83D\uDE80y';

// ES5: \uD83D\uDE80 are (incorrectly) reversed
console.log(str.split('').reverse().join(''));
    // 'y\uDE80\uD83Dx'

// ES6: order of \uD83D\uDE80 is preserved
console.log([...str].reverse().join(''));
    // 'y\uD83D\uDE80x'
```

在火狐浏览器中的翻转结果：



> 遗留问题：合并标记 一个合并标记就是两个 Unicode 码点的序列，用于展示单个符号。我在这里展示的 ES6 翻转非 BMP 码点的方法，不适用于合并标记。对于合并标记的翻转，你需要一个库，例如 Mathias Bynens 的 Esrever 。

6.5 码点的数字值

新方法 `codePointAt()` 返回字符串中指定位置字符的数字值：

```
const str = 'x\uD83D\uDE80y';
console.log(str.codePointAt(0).toString(16)); // 78
console.log(str.codePointAt(1).toString(16)); // 1f680
console.log(str.codePointAt(3).toString(16)); // 79
```

该方法在字符串遍历中也能正确执行：

```
for (const ch of 'x\uD83D\uDE80y') {
    console.log(ch.codePointAt(0).toString(16));
}
// Output:
// 78
// 1f680
// 79
```

与 `codePointAt()` 对应的方法是 `String.fromCodePoint()` ：

```
> String.fromCodePoint(0x78, 0x1f680, 0x79) === 'x\uD83D\uDE80y'
true
```

6.6 检查子串

有三个方法用于检查某个字符串是否是另一个字符串的子串：

```
> 'hello'.startsWith('hell')
true
> 'hello'.endsWith('ello')
true
> 'hello'.includes('ell')
true
```

每个方法都有可选的第二个参数，用于指定字符串搜索的开始或者结束位置：

```
> 'hello'.startsWith('ello', 1)
true
> 'hello'.endsWith('hell', 4)
true

> 'hello'.includes('ell', 1)
true
> 'hello'.includes('ell', 2)
false
```

6.7 重复字符串

`repeat()` 方法用于生成重复字符串：

```
> 'doo '.repeat(3)
'doo doo doo '
```

6.8 用正则表达式作为参数的字符串方法

在 ES6 中，四个用正则表达式作为参数的字符串方法做的事情相当少，它们主要调用参数上面的方法：

- `String.prototype.match(regexp)` 调用 `regexp[Symbol.match](this)` 。
- `String.prototype.replace(searchValue, replaceValue)` 调用 `searchValue[Symbol.replace](this, replaceValue)` 。
- `String.prototype.search(regexp)` 调用 `regexp[Symbol.search](this)` 。
- `String.prototype.split(separator, limit)` 调用 `separator[Symbol.split](this, limit)` 。

这些参数根本不需要是正则表达式，任何带有相应方法的对象都可以作为参数。

6.9 备忘：新的字符串方法

标签模板：

- String.raw(callSite, ...substitutions) : string

  用于获取“原始”字符串内容的模板标签（反斜杠不再是转义字符）：

  ```
  > String.raw`\` === '\\'
  true
  ```

  更多内容可阅读关于模板字面量的章节。

Unicode 和码点：

- String.fromCodePoint(...codePoints : number[]) : string

  将数字值转换成 Unicode 码点字，然后返回由码点构成的字符串。

- String.prototype.codePointAt(pos) : number

  返回在从位置 `pos` 处开始的码点的数字值（由一个或者两个 JavaScript 字符组成）。

- String.prototype.normalize(form? : string) : string

  不同的码点组合可能看起来是一样的。Unicode 标准化 将它们修正为相同的标准值。这对相等比较和字符串搜索很有帮助。对于一般的文本，建议使用 `NFC` 形式。

查找字符串：

- String.prototype.startsWith(searchString, position=0) : boolean

  `position` 参数指定了字符串的开始搜索位置。

- String.prototype.endsWith(searchString, endPosition=searchString.length) : boolean

  `endPosition` 指定了字符串的结束搜索位置。

- String.prototype.includes(searchString, position=0) : boolean

  从字符串 `position` 位置开始搜索，是否包含 `searchString` 子串。

重复字符串：

- String.prototype.repeat(count) : string

  返回重复指定次数的字符串。

7 Symbol

Symbol 是 ECMAScript 6 中一个新的原始类型，本章讲解了它的工作原理。

7.1 概览

7.1.1 用途1：唯一属性键

Symbol 主要用作唯一属性键 - 一个 symbol 对象不会与任何其他属性键（另一个 symbol 对象或者字符串）冲突。例如，你可以将 `Symbol.iterator` 作为某个对象的键（键值是一个方法），使其变得可迭代（可以通过 `for-of` 或者其他语言机制来迭代，更多相关内容可以在有关迭代的章节找到）：

```
const iterableObject = {
    [Symbol.iterator]() { // (A)
        const data = ['hello', 'world'];
        let index = 0;
        return {
            next() {
                if (index < data.length) {
                    return { value: data[index++] };
                } else {
                    return { done: true };
                }
            }
        };
    }
}
for (const x of iterableObject) {
    console.log(x);
}
// Output:
// hello
// world
```

在行 A 处， symbol 对象用作键，键值是一个方法。这个唯一的键使得该对象可迭代，可用于 `for-of` 循环。

7.1.2 用途2：用常量代表特殊的含义

在 ECMAScript 5 中，你可能会使用字符串来表示一些特殊的含义，比如颜色。在 ES6 中，可以使用 symbol ，因为 symbol 总是唯一的：

```
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');

function getComplement(color) {
    switch (color) {
        case COLOR_RED:
            return COLOR_GREEN;
        case COLOR_ORANGE:
            return COLOR_BLUE;
        case COLOR_YELLOW:
            return COLOR_VIOLET;
        case COLOR_GREEN:
            return COLOR_RED;
        case COLOR_BLUE:
            return COLOR_ORANGE;
        case COLOR_VIOLET:
            return COLOR_YELLOW;
        default:
            throw new Exception('Unknown color: '+color);
    }
}
```

7.1.3 陷阱：不能将 symbol 强制转换成字符串

将 symbol 强制（隐式地）转换成字符串会抛出异常：

```
const sym = Symbol('desc');

const str1 = '' + sym; // TypeError
const str2 = `${sym}`; // TypeError
```

只能通过显示的方式转换：

```
const str2 = String(sym); // 'Symbol(desc)'
const str3 = sym.toString(); // 'Symbol(desc)'
```

禁止强制转换能够避免一些错误，但是也使得 symbol 的使用变得复杂起来。

7.1.4 哪些跟属性相关的操作能感知到 symbol 键？

下面的操作能感知到 symbol 键：

- `Reflect.ownKeys()`
- 通过 `[]` 访问属性
- `Object.assign()`

下面的操作会忽略掉 symbol 键：

- `Object.keys()`
- `Object.getOwnPropertyNames()`
- `for-in` 循环

7.2 新的原始类型

ECMAScript 6 引入了一种新的原始类型： symbol 。它能被用作唯一标识的 ID 。通过工厂方法 Symbol() 来创建 symbol 对象（这跟 `String` 方法返回字符串很类似）：

```
const symbol1 = Symbol();
```

`Symbol()` 有一个可选的字符串参数，用作新创建的 symbol 对象的一个描述文本。该文本在 symbol 被转换为字符串（通过 `toString()` 或者 `String()` ）的时候用到：

```
> const symbol2 = Symbol('symbol2');
> String(symbol2)
'Symbol(symbol2)'
```

`Symbol()` 方法返回的每一个 symbol 实例都是唯一的，每一个 symbol 对象都有自己的标识：

```
> Symbol() === Symbol()
false
```

如果你用 `typeof` 来检测 symbol 对象，会发现 symbol 是原始类型，因为返回值是一种新的特定于 symbol 对象的类型：

```
> typeof Symbol()
'symbol'
```

7.2.1 用作属性键的 symbol

symbol 可以用作属性键：

```
const MY_KEY = Symbol();
const obj = {};

obj[MY_KEY] = 123;
console.log(obj[MY_KEY]); // 123
```

类和对象字面量有一个叫做计算属性键的特性：你可以用一个表达式来指定属性键，将表达式放置在中括号里面。如下的对象字面量，使用了计算属性键的方式，让 `MY_KEY` 成为了键。

```
const MY_KEY = Symbol();
const obj = {
    [MY_KEY]: 123
};
```

计算键同样适用于方法定义：

```
const FOO = Symbol();
const obj = {
    [FOO]() {
        return 'bar';
    }
};
console.log(obj[FOO]()); // bar
```

7.2.2 枚举自有属性键

鉴于现在有一种新的可以用作属性键的值了，那么 ECMAScript 6 中就产生了新的术语：

- 属性键要么是字符串，要么是 symbol 对象。
- 字符串形式的属性键叫做属性名字。
- symbol 对象形式的属性键叫做属性 *symbol* 。

我们先创建一个对象，然后用这个对象来验证新的自有属性枚举 API 。

```
const obj = {
    [Symbol('my_key')]: 1,
    enum: 2,
    nonEnum: 3
};
Object.defineProperty(obj,
    'nonEnum', { enumerable: false });
```

`Object.getOwnPropertyNames()` 忽略 symbol 形式的属性键：

```
> Object.getOwnPropertyNames(obj)
['enum', 'nonEnum']
```

`Object.getOwnPropertySymbols()` 忽略字符串形式的属性键：

```
> Object.getOwnPropertySymbols(obj)
[Symbol(my_key)]
```

`Reflect.ownKeys()` 返回所有类型的键：

```
> Reflect.ownKeys(obj)
[Symbol(my_key), 'enum', 'nonEnum']
```

`Object.keys()` 仅返回可枚举的字符串属性：

```
> Object.keys(obj)
['enum']
```

`Object.keys` 这个名字与新术语（仅列举字符串类型的键）冲突了。现在，`Object.names` 或者 `Object.getEnumerableOwnPropertyNames` 会是更好的选择。

7.3 使用 Symbol 来表达一些概念

在 ECMAScript 5 中，一种常用的表达特殊含义（想想枚举常量）是通过字符串来实现的。例如：

```
var COLOR_RED    = 'Red';
var COLOR_ORANGE = 'Orange';
var COLOR_YELLOW = 'Yellow';
var COLOR_GREEN  = 'Green';
var COLOR_BLUE   = 'Blue';
var COLOR_VIOLET = 'Violet';
```

然而，字符串并不是我们想象中的那么具有唯一性。为什么呢？请看如下函数。

```
function getComplement(color) {
    switch (color) {
        case COLOR_RED:
            return COLOR_GREEN;
        case COLOR_ORANGE:
            return COLOR_BLUE;
        case COLOR_YELLOW:
            return COLOR_VIOLET;
        case COLOR_GREEN:
            return COLOR_RED;
        case COLOR_BLUE:
            return COLOR_ORANGE;
        case COLOR_VIOLET:
            return COLOR_YELLOW;
        default:
            throw new Exception('Unknown color: '+color);
    }
}
```

很明显，你可以使用任何表达式作为 `switch case` 中的表达式，并不局限于某种形式。例如：

```
function isThree(x) {
    switch (x) {
        case 1 + 1 + 1:
            return true;
        default:
            return false;
    }
}
```

我们使用了 `switch` 提供的便利，用常量来指代颜色（ `COLOR_RED` 等等），而不是使用硬编码的方式（' `RED` ' 等）。

有趣的是，即便我们使用了常量，还是会产生混淆。例如，某个人可能随手定义一个常量：

```
var MOOD_BLUE = 'BLUE';
```

现在， `BLUE` 值再也不是唯一的了， `MOOD_BLUE` 与其冲突了。如果将它作为 `getComplement()` 的参数，原本期望能抛出异常，结果却返回了 `'ORANGE'` 。

让我们使用 symbol 来解决这个问题。现在，我们同样能够使用 ES6 的 `const` 特性，这样就可以声明真正的常量了（不能改变常量的指向，但是值本身可能是可变的）。

```
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');
```

Symbol 返回的每一个值都是唯一的，这就是为什么没有其他的值可以和 `BLUE` 发生冲突的原因。有趣的是，如果我们使用 symbol 对象替换字符串，并不需要改动 `getComplement()` 函数，这表明两种方案的代码是多么的相似。

7.4 Symbol 作为属性键

创建永远不会冲突的键，在以下两种场景中非常有用：

- 在继承树中定义非公开的属性。
- 保证子层属性不会和父层属性冲突。

7.4.1 Symbol 用作非公开属性键

无论何种形式的 JavaScript 继承（例如通过类，通过 mixin ，或者就是单纯的原型的方式），都会存在两种类型的属性：

- 公开属性 能被客户代码访问。
- 私有属性 在组成继承结构的代码片段（例如类、 mixin 或者对象）内部使用。（受保护的属性 被若干个代码片段共享，和私有属性面临同样的问题。）

为了可用性的缘故，公开属性通常用字符串作为键。但是，对于字符串作为键的私有属性，意外的命名冲突可能会导致一些问题。因此， symbol 是一个好的选择。例如，在下面的代码中， symbol 用于私有属性 `_counter` 和 `_action` 。

```
const _counter = Symbol('counter');
const _action = Symbol('action');
class Countdown {
    constructor(counter, action) {
        this[_counter] = counter;
        this[_action] = action;
    }
    dec() {
        let counter = this[_counter];
        if (counter < 1) return;
        counter--;
        this[_counter] = counter;
        if (counter === 0) {
            this[_action]();
        }
    }
}
```

注意 Symbol 仅会使你免于命名冲突，并不会阻止未授权的属性访问，因为可以通过 `Reflect.ownKeys()` 找出对象所有自有属性键，包括 symbol 。如果你想对属性做保护，可以选择在这一节中讲述的方法之一：类的私有数据。

7.4.2 Symbol 用作元级属性键

相对于“普通的”属性键，Symbol 的唯一性使其在一个不同的纬度成为理想的公开属性，因为元级键和普通键不能冲突。举个元级键的例子，对象可以实现一些特定的方法，从而自定义某个库如何处理该对象。使用 Symbol 键可以避免第三方库错误地将普通方法当成自定义方法。

ECMAScript 6 中的可迭代性就属于这种自定义方法的场景。如果一个对象有一个键为 `Symbol.iterator` 的方法，那么它就是可迭代的。在下面的例子中，`obj` 是可迭代的。

```
const obj = {
    data: [ 'hello', 'world' ],
    [Symbol.iterator]() {
        const self = this;
        let index = 0;
        return {
            next() {
                if (index < self.data.length) {
                    return {
                        value: self.data[index++]
                    };
                } else {
                    return { done: true };
                }
            }
        };
    }
};
```

`obj` 的可迭代性使其可用于 `for-of` 循环，以及类似的 JavaScript 特性中：

```
for (const x of obj) {
    console.log(x);
}

// Output:
// hello
// world
```

7.4.3 JavaScript 标准库命名冲突举例

或许你觉得命名冲突不要紧，下面有三个例子，展示了在 JavaScript 标准库演化过程中命名冲突带来的问题：

- 在引入新方法 `Array.prototype.values()` 的时候，破坏了已有的 `with` 与数组结合使用的代码，它会在外面的作用域中隐藏一个 `values` 变量（bug report 1，bug report 2）。因此，引入了一种隐藏属性的机制(System.unscopables)。
- `String.prototype.contains` 和 MooTools 中添加的一个方法冲突了，不

得不重命名成 `String.prototype.includes` （ bug report ）。
- ES2016 中的 `Array.prototype.contains` 方法也和 MooTools 中添加的方法冲突了，不得不重命名成 `Array.prototype.includes` （ bug report ）。

相比之下，通过属性键 `System.iterator` 给一个对象增加可迭代性就不会引起问题，因为该键并不会和任何键冲突。

上述例子展示了成为一门 web 开发语言意味着什么：向后兼容是至关重要的，这就是为什么在改进语言的时候，偶尔妥协是必须的。这种向后兼容的好处就是，改进老的代码库很轻松，因为新的 ECMAScript 版本绝不会（好吧，是几乎不会）破坏这些老代码。

7.5 Symbol 转换成其他原始类型

下面的表格展示了 Symbol 在显示地或者隐式地转换成其他原始类型的时候会发生什么：

转换成	显示转换	隐式转换
boolean	Boolean(sym) → OK	!sym → OK
number	Number(sym) → TypeError	sym*2 → TypeError
string	String(sym) → OK	"+sym → TypeError
	sym.toString() → OK	`${sym}` → TypeError

7.5.1 当心：隐式转换成字符串

禁止隐式转换成字符串会很容易导致下面的错误：

```
const sym = Symbol();

console.log('A symbol: '+sym); // TypeError
console.log(`A symbol: ${sym}`); // TypeError
```

要修复这些问题，需要显示转换成字符串：

```
console.log('A symbol: '+String(sym)); // OK
console.log(`A symbol: ${String(sym)}`); // OK
```

7.5.2 理解隐式转换规则

通常会禁止 Symbol 的隐式转换。本节讲解了为什么会这样。

7.5.2.1 允许真值检查

隐式转换成布尔值是始终可以的，主要用于在 if 语句中检查真值情况以及其他一些场景：

```
if (value) { ··· }

param = param || 0;
```

7.5.2.2 意外地将 Symbol 用作属性键

Symbol 是一种特殊的属性键，这就是为什么需要避免意外地将其转换成字符串（字符串是另一种类型的属性键）。在使用一些操作符来计算属性名的时候，这种意外错误就可能发生：

```
myObject['__' + value]
```

这就是为啥 `value` 是 Symbol 的时候会抛出 `TypeError` 错误。

7.5.2.3 意外地将 Symbol 用作数组索引

你肯定也不期望意外地将 Symbol 用作数组索引。下面的代码展示了当 `value` 是 Symbol 的时候，这种错误就会发生：

```
myArray[1 + value]
```

这就是为什么在这种场景中，该操作会抛出错误。

7.5.3 规范中的显示转换和隐式转换

7.5.3.1 转换成布尔类型

可以使用 Boolean() 显示地将 Symbol 转换成布尔类型。对于 Symbol 类型，会始终返回 `true` ：

```
> const sym = Symbol('hello');
> Boolean(sym)
true
```

`Boolean()` 通过内部的 ToBoolean() 操作计算出最终结果。对于 Symbol 和其他真值，该操作返回 `true` 。

隐式转换同样使用 `ToBoolean()` ：

```
> !sym
false
```

7.5.3.2 转换成数值类型

使用 Number() 将 Symbol 显示地转换成数值类型：

```
> const sym = Symbol('hello');
> Number(sym)
TypeError: can't convert symbol to number
```

`Number()` 通过内部的 ToNumber() 操作计算出最终结果，对于 Symbol ，该操作会抛出 `TypeError` 。

隐式转换也会使用 [ToNumber()] ：

```
> +sym
TypeError: can't convert symbol to number
```

7.5.3.3 转换成字符串

使用 String() 将 Symbol 转换成字符串：

```
> const sym = Symbol('hello');
> String(sym)
'Symbol(hello)'
```

如果 `String()` 的参数是 Symbol ，那么该方法会自己将其转换成字符串，然后返回用创建 Symbol 的时候传入的描述字符串（使用 `Symbol()` 包裹起来）。如果没有提供描述字符串，就会使用空字符串：

```
> String(Symbol())
'Symbol()'
```

`toString()` 方法返回和 `String()` 相同的结果，但是都不会调用对方，它们都会调用内部的操作 SymbolDescriptiveString() 。

```
> Symbol('hello').toString()
'Symbol(hello)'
```

隐式转换是通过内部的 ToString() 操作来处理的，对于 Symbol ，会抛出 `TypeError` 。 `Number.parseInt()` 方法会隐式地将参数转换成字符串：

```
> Number.parseInt(Symbol())
TypeError: can't convert symbol to string
```

7.5.3.4 不允许：使用双目加号操作符（+）转换

加号操作符的处理过程是这样的：

- 将两边的操作数转换成原始类型。
- 如果某一个操作数是字符串，那么另一个操作数会被隐式转换成字符串（使用 `ToString()` ），然后将它们串联起来，返回结果。
- 否则，将两个操作数都转换成数值类型，然后相加，返回结果。

隐式转换成字符串和数值都会抛出异常，这意味着不能（直接）将加号运算符用于 Symbol 。

```
> '' + Symbol()
TypeError: can't convert symbol to string
> 1 + Symbol()
TypeError: can't convert symbol to number
```

7.6 JSON 与 Symbol

7.6.1 用 JSON.stringify() 生成 JSON

`JSON.stringify()` 将 JavaScript 数据转换成 JSON 字符串。有一个预处理的步骤可以自定义转换：传入一个转换函数，可以将 JavaScript 数据中的任何数据转换成另一种数据。这意味着可以将不兼容 JSON 的值（比如 Symbol 和日期）转换为 JSON 兼容的值（比如字符串）。 `JSON.parse()` 使用类似的过程解析 JSON 字符串。

但是， `stringify` 会忽略掉非字符串类型的属性键，所以该方法仅对作为属性值的 Symbol 有效。例如，像这样：

```
function symbolReplacer(key, value) {
    if (typeof value === 'symbol') {
        return '@@' + Symbol.keyFor(value) + '@@';
    }
    return value;
}
const MY_SYMBOL = Symbol.for('http://example.com/my_symbol');
const obj = { myKey: MY_SYMBOL };

const str = JSON.stringify(obj, symbolReplacer);
console.log(str);
    // {"myKey":"@@http://example.com/my_symbol@@"}
```

Symbol 被编码成以'@@'为前缀和后缀的字符串。注意，只有通过 `Symbol.for()` 创建的 Symbol 才会有这样的键。

7.6.2 用 JSON.parse() 解析 JSON

`JSON.parse()` 将 JSON 字符串转换成 JavaScript 数据。有一个后处理步骤可以自定义转换：传入一个转换函数，所谓的复活器（*reviver*），可以将 JSON 字符串中的任何原始串成其它的数据。这样就可以将指定格式的 JSON 数据转换成非 JSON 规范支持的数据了。如下所示：

```
const REGEX_SYMBOL_STRING = /^@@(.*)@@$/;
function symbolReviver(key, value) {
    if (typeof value === 'string') {
        const match = REGEX_SYMBOL_STRING.exec(value);
        if (match) {
            const symbolKey = match[1];
            return Symbol.for(symbolKey);
        }
    }
    return value;
}

const parsed = JSON.parse(str, symbolReviver);
console.log(parse);
```

以‘@@’为前缀和后缀的字符串通过解析中间的 Symbol 键被转换成 Symbol 。

7.7 Symbol 包装对象

虽然其它所有原始类型都有字面量形式的值，但是要创建 Symbol 对象，还是得调用 `Symbol` 函数。因此，有时候很容易把 `Symbol` 函数当成构造函数使用。虽然这会返回 Symbol 的一个实例，但是并没有什么用。所以，如果把 Symbol 函数当成构造函数使用的话，会抛出异常：

```
> new Symbol()
TypeError: Symbol is not a constructor
```

有一种创建包装对象的方法：将 `Object` 当做一个普通函数调用，会将所有的值（包括 Symbol）转换成对象。

```
> const sym = Symbol();
> typeof sym
'symbol'

> const wrapper = Object(sym);
> typeof wrapper
'object'
> wrapper instanceof Symbol
true
```

7.7.1 使用 `[]` 和包装对象键访问属性

用来访问属性的方括号操作符 `[]` 会将字符串包装对象和 Symbol 包装对象解包装。我们可以使用如下的对象来检测这个现象。

```
const sym = Symbol('yes');
const obj = {
    [sym]: 'a',
    str: 'b',
};
```

使用交互模式：

```
> const wrappedSymbol = Object(sym);
> typeof wrappedSymbol
'object'
> obj[wrappedSymbol]
'a'

> const wrappedString = new String('str');
> typeof wrappedString
'object'
> obj[wrappedString]
'b'
```

7.7.1.1 规范中关于属性访问的说明

获取和设置属性的操作是通过内部的 ToPropertyKey() 操作实现的，其工作过程如下：

- 用 ToPrimitive() 把操作数转换成原始值类型（优先选用 `string` 类型）：
    - 如果是原始值，就返回自身。
    - 对于非 Symbol 对象，如果 `toString()` 返回原始值，就使用 `toString()` 转换；否则，如果 `valueOf()` 返回原始值，就使用 `valueOf()` ；否则，抛出 `TypeError` 错误。
    - Symbol 对象是一个例外：它们会被转换成原始的 Symbol （解包装）。
- 如果转换结果是 Symbol 原始值，直接返回该值。
- 否则，通过 ToString() 强制将结果转换成字符串。

9 变量与作用域

本章将介绍在 ECMAScript 6 中如何处理变量与作用域。

9.1 概览

ES6 新增了两个定义变量的关键字：`let` 与 `const`，它们几乎取代了 ES5 定义变量的方式：`var`。

9.1.1 `let`

`let` 语法上非常类似于 `var`，但定义的变量是语句块级作用域，只存在于当前的语句块中。`var` 拥有函数作用域。

在如下的代码中，`let` 定义的变量 `tmp` 只存在于行 A 开始的语句块中：

```
function order(x, y) {
    if (x > y) { // (A)
        let tmp = x;
        x = y;
        y = tmp;
    }
    console.log(tmp === x); // ReferenceError: tmp is not defined

    return [x, y];
}
```

9.1.2 `const`

`const` 和 `let` 类似，但是定义变量时必须初始化值，并且是只读的。

```
const foo;
    // SyntaxError: missing = in const declaration

const bar = 123;
bar = 456;
    // TypeError: `bar` is read-only
```

9.1.3 定义变量的方式

下面的表格对比了在 ES6 中定义变量的 6 种方式：

	Hoisting	Scope	Creates global properties
var	Declaration	Function	Yes
let	Temporal dead zone	Block	No
const	Temporal dead zone	Block	No
function	Complete	Block	Yes
class	No	Block	No
import	Complete	Module-global	No

9.2 通过 `let` 和 `const` 实现块级作用域

`let` 和 `const` 创建的变量都是块级作用域：它们只存在于包围它们的最深的代码块中。以下的代码表明了 `let` 声明的变量 `tmp` 仅仅存在于 `if` 声明内部。

```
function func() {
    if (true) {
        let tmp = 123;
    }
    console.log(tmp); // ReferenceError: tmp is not defined
}
```

相比之下， `var` 声明的变量是函数域：

```
function func() {
    if (true) {
        var tmp = 123;
    }
    console.log(tmp); // 123
}
```

块级作用域意味着你能在函数中重复声明变量（shadow variables）：

```
function func() {
  let foo = 5;
  if (···) {
     let foo = 10; // shadows outer `foo`
     console.log(foo); // 10
  }
  console.log(foo); // 5
}
```

9.3 `const` 创建不可变的变量

`let` 创建的变量是可变的：

```
let foo = 'abc';
foo = 'def';
console.log(foo); // def
```

常量通过 `const` 创建，是不可变的 - 不能重新赋值：

```
const foo = 'abc';
foo = 'def'; // TypeError
```

9.3.1 陷阱： `const` 不能确保值不可变

`const` 仅仅意味着变量总是有相同的值，但是不能确保可变的值不可变。例如，`obj` 是一个常量，但是它指向的值是可变的 - 我们能增加一个属性给它：

```
const obj = {};
obj.prop = 123;
console.log(obj.prop); // 123
```

然而，我们不能重新分配一个不同的值给 `obj` ：

```
obj = {}; // TypeError
```

如果你想让 `obj` 不可变，你必须做一些处理，例如冻结对象：

```
const obj = Object.freeze({});
obj.prop = 123; // TypeError
```

9.3.2 循环体中的 `const`

`const` 变量一旦被创建， 它就不能被改变。 但是这并不是意味着你不能通过循环重新进入它的作用域并且赋值重新执行：

```
function logArgs(...args) {
    for (let [index, elem] of args.entries()) {
        const message = index + '. ' + elem;
        console.log(message);
    }
}
logArgs('Hello', 'everyone');

// Output:
// 0. Hello
// 1. everyone
```

9.4 暂时性死区（temporal dead zone）

`let` 或 `const` 声明的变量拥有暂时性死区（TDZ）：当进入它的作用域，它不能被访问（获取或设置）直到执行到达声明。

首先看看不具有暂时性死区的 `var`：

- 当进入 `var` 变量的作用域（包围它的函数），立即为它创建（绑定）存储空间。变量会立即被初始化并赋值为 `undefined`。
- 当执行到变量声明的时候，如果变量定义了值则会被赋值。

通过 `let` 声明的变量拥有暂时性死区，生命周期如下：

- 当进入 `let` 变量的作用域（包围它的语法块），立即为它创建（绑定）存储空间。此时变量仍是未初始化的。
- 获取或设置未初始化的变量将抛出异常 `ReferenceError`。
- 当执行到变量声明的时候，如果变量定义了值则会被赋值。如果没有定义值，则赋值为 `undefined`。

`const` 工作方式与 `let` 类似，但是定义的时候必须赋值并且不能改变。

在 TDZ 内部，如果获取或设置变量将抛出异常：

```
if (true) { // enter new scope, TDZ starts
    // Uninitialized binding for `tmp` is created

    tmp = 'abc'; // ReferenceError
    console.log(tmp); // ReferenceError

    let tmp; // TDZ ends, `tmp` is initialized with `undefined`
    console.log(tmp); // undefined

    tmp = 123;
    console.log(tmp); // 123
}
```

下面的示例将演示死区（dead zone）是真正短暂的（基于时间）和不受空间条件限制（基于位置）

```
if (true) { // enter new scope, TDZ starts
    const func = function () {
        console.log(myVar); // OK!
    };

    // Here we are within the TDZ and
    // accessing `myVar` would cause a `ReferenceError`

    let myVar = 3; // TDZ ends
    func(); // called outside TDZ
}
```

9.4.1 `typeof` 与暂时性死区

变量在暂时性死区无法被访问，所以无法对它使用 `typeof` ：

```
if (true) {
    console.log(typeof tmp); // ReferenceError
    let tmp;
}
```

我不认为这对大多数程序员来说是个问题：用这种方式检查变量是否存在的主要存在于库中 （特别是 polyfills）。如果你需要检查一个变量是否存在，你可以通过其他的方式（例如通过 `window` ）。最后，从长远来看模块会减少这种类型的需求。

9.5 循环头中的 `let` 和 `const`

在以下循环头中允许声明变量：

- `for`
- `for-in`
- `for-of`

声明一个变量，你可以使用 `var`，`let` 或者 `const`。每一种方式有不一样的效果，下面我将进行解释。

9.5.1 `for` 循环

在 `for` 循环头中使用 `var` 创建的变量只创建单个绑定（binding）：

```
let arr = [];
for (var i=0; i < 3; i++) {
    arr.push(() => i);
}
arr.map(x => x()); // [3,3,3]
```

在代码块中的三个箭头函数中的 `i` 都指向了同一个 binding，这就是为什么他们都返回相同的值。

如果你使用 `let` 声明变量，循环的每一次都会创建一个 binding ：

```
let arr = [];
for (let i=0; i < 3; i++) {
    arr.push(() => i);
}
arr.map(x => x()); // [0,1,2]
```

这一次，每个 `i` 指向一个循环体中的 binding 并且保持当时的值。因此，每个箭头函数都返回不同的值。

`const` 工作方式与 `var` 类似，但是不能改变 `const` 声明变量时初始化的值。

刚开始也许会觉得在每个循环中获取新的 binding 似乎很奇怪，但是当你使用循环创建指向循环变量的函数（例如事件处理的回调）时它非常有用。

> **for** 循环：规范中的迭代绑定
> The evaluation of the `for` 循环 在第二例介绍 `var` 并且在第三例介绍 `let/const` 。仅仅 `let` 定义的变量被加到 `perIterationLets` 列表（步骤 9），作为传递给 ForBodyEvaluation() 的第一个之后的参数 `perIterationBindings` 。

9.5.2 `for-of` 循环和 `for-in` 循环

In a `for-of` loop, `var` creates a single binding:

在 `for-of` 循环头中使用 `var` 创建的变量只创建单个绑定（binding）：

```
let arr = [];
for (var i of [0, 1, 2]) {
    arr.push(() => i);
}
arr.map(x => x()); // [2,2,2]
```

`let` 在循环的每一次都会创建一个 binding ：

```
let arr = [];
for (let i of [0, 1, 2]) {
    arr.push(() => i);
}
arr.map(x => x()); // [0,1,2]
```

`const` 也会在循环的每一次都会创建一个 binding ，但是创建的 bindings 是不可变的。

`for-in` 循环与 `for-of` 工作方式类似.

> **for-of** 循环：规范中的迭代绑定
> 在 ForIn/OfBodyEvaluation 中介绍了 `for-of` 的迭代绑定。在步骤 5.b 中，创建一个新的环境，并且绑定通过 BindingInstantiation 被增加。当前的迭代值存入变量 `nextValue` ，并且在如下两种方式中用于初始化绑定：
>
> - 定义单个变量（步骤 5.h.i）： 通过 InitializeReferencedBinding
> - 解构（步骤 5.i.iii）：通过 BindingInitialization 中的一种（ForDeclaration），调用 BindingInitialization 的另一种（BindingPattern）。

9.6 参数

9.6.1 参数与本地可用变量

如果你使用 `let` 定义一个与参数名相同的变量，会抛出静态错误：

```
function func(arg) {
    let arg; // static error: duplicate declaration of `arg`
}
```

如果在块中定义：

```
function func(arg) {
    {
        let arg; // shadows parameter `arg`
    }
}
```

相比之下，`var` 定义与参数名相同的变量不会做任何处理。

```
function func(arg) {
    var arg; // does nothing
}
function func(arg) {
    {
        // We are still in same `var` scope as `arg`
        var arg; // does nothing
    }
}
```

9.6.2 参数默认值与暂时性死区

如果参数有默认值，也会处于暂时性死区：

```
// OK: `y` accesses `x` after it has been declared
function foo(x=1, y=x) {
    return [x, y];
}
foo(); // [1,1]

// Exception: `x` tries to access `y` within TDZ
function bar(x=y, y=2) {
    return [x, y];
}
bar(); // ReferenceError
```

9.6.3 参数默认值无法访问函数内作用域

参数默认值的作用域被方法主体分离。那就意味着方法或者函数的参数默认值不能访问方法块内本地可用变量：

```
let foo = 'outer';
function bar(func = x => foo) {
    let foo = 'inner';
    console.log(func()); // outer
}
bar();
```

9.7 全局对象

JavaScript 中的全局对象（浏览器中为 `window`，Node.js 中为 `global`）相对于提供的功能来说更算是一个 bug，特别是在性能方面。这就是为什么 ES6 这样规定：

- 以下的命令可以声明全局变量，属于全局对象的属性：
  - `var`
  - `function`
- 但下面的命令声明的全局变量，不属于全局对象的属性：
  - `let`
  - `const`
  - `class`

9.8 函数声明和类 声明

函数 声明......

- 块级作用域，类似 `let` 。
- 创建全局对象的属性（在全局作用域中），类似 `var` 。
- 变量提升：在作用域中的所有函数声明，都会在作用域的最顶部创建。

以下的代码演示了函数声明的变量提升：

```
{ // Enter a new scope

    console.log(foo()); // OK, due to hoisting
    function foo() {
        return 'hello';
    }
}
```

类声明......

- 块级作用域
- 不创建全局对象的属性
- 变量不提升

类变量不提升可能是令人惊讶的，因为在本质上它们创建了函数。这种行为的理由是，继承的值是通过表达式定义，并且这些表达式必须中适当的时间执行。

```
{ // Enter a new scope

    const identity = x => x;

    // Here we are in the temporal dead zone of `MyClass`
    let inst = new MyClass(); // ReferenceError

    // Note the expression in the `extends` clause
    class MyClass extends identity(Object) {
    }
}
```

9.9 代码风格： `var` VS `let` VS `const`

`var` 能做一件 `let` 和 `const` 不能做的事：通过它声明的变量成为全局对象的属性。但是可以通过分配给 `window` （浏览器中）和 `global` （Node.js 中）实现同样的效果。因此，我建议总是使用 `let` 和 `const` 取代 `var` 。

`const` 用于不可变变量：

```
**// Primitive values are immutable
const PUBLIC_SYMBOL = Symbol();
const MAX_ENTRIES = 1000;

// Some objects are immutable
const EMPTY_ARRAY = Object.freeze([]);
Use let for mutable things:

// A primitive whose value changes
let counter = 0;
counter++;

// A mutable object
let obj = {};
obj.foo = 123;// Primitive values are immutable
const PUBLIC_SYMBOL = Symbol();
const MAX_ENTRIES = 1000;

// Some objects are immutable
const EMPTY_ARRAY = Object.freeze([]);
Use let for mutable things:

// A primitive whose value changes
let counter = 0;
counter++;

// A mutable object
let obj = {};
obj.foo = 123;**
```

这并不是一个艰难的抉择，并且使用 `const` 声明的可变的对象也没有问题。

10 解构

ECMAScript 6 （ ES6 ）支持解构，一种从对象和数组中方便地提取数据的方式。本章叙述了这如何工作，并且给出一些例子展示它的用处。

10.1 概览

在接收数据的地方（比如赋值的左边），解构使你使用模式去获取部分数据。

下面的代码是解构的一个例子：

```
let obj = { first: 'Jane', last: 'Doe' };
let { first: f, last: l } = obj; // (A)
    // f = 'Jane'; l = 'Doe'
```

在行 A 解构了 `obj` ：通过左边的模式，运用赋值操作符（=）从里面获取数据，并将数据赋值给变量 `f` 和 `l` 。这些变量事先自动声明好，因为该行以 `let` 开始。

也可以解构数组：

```
let [x, y] = ['a', 'b']; // x = 'a'; y = 'b'
```

解构可以用于下列情形：

```
// Variable declarations:
let [x] = ['a'];
const [x] = ['a'];
var [x] = ['a'];

// Assignments:
[x] = ['a'];

// Parameter definitions:
function f([x]) { ··· }
f(['a']);
```

也可以在一个 `for-of` 循环中解构：

```
// Handled like a variable declaration:
for (let [k,v] of arr.entries()) ···

// Handled like an assignment
for ({name: n, age: a} of arr) ···
```

10.2 背景：构造数据（对象和数组字面量）和解构数据

为了完全理解解构是什么，首先学习更广泛的上下文知识。JavaScript 有构造数据的操作：

```
let obj = {};
obj.first = 'Jane';
obj.last = 'Doe';
```

也有获取数据的操作：

```
let f = obj.first;
let l = obj.last;
```

注意，我们使用了与构造数据时一样的语法。

有更漂亮的语法来构造数据 - 对象字面量：

```
let obj = { first: 'Jane', last: 'Doe' };
```

ECMAScript 6 中的解构使用了相同的语法来获取数据，这被称为一种对象模式：

```
let { first: f, last: l } = obj;
```

就像对象字面量让我们一次性创建多个属性一样，对象模式让我们一次性获取多个属性值。

也可以通过模式解构数组：

```
let [x, y] = ['a', 'b']; // x = 'a'; y = 'b'
```

10.3 模式

下面的两部分包含在解构过程当中：

- 解构源：被解构的数据。例如，解构赋值右侧的部分。
- 解构目标：用于解构的模式。例如，解构赋值左侧的部分。

解构目标是下列三种模式之一：

- 赋值目标。例如： `x`
    - 在变量声明和参数定义中，仅允许指向变量。在解构赋值中，就有更多的选择了，后面再讲解。
- 对象模式。例如： `{ first: «pattern», last: «pattern» }`
    - 对象模式的组成部分是属性，属性值再一次成为模式（递归地）。
- 数组模式。例如： [«pattern», «pattern»]
    - 数组模式的组成部分是元素，元素再一次成为模式（递归地）。

这意味着可以任意深地内嵌模式：

```
let obj = { a: [{ foo: 123, bar: 'abc' }, {}], b: true };
let { a: [{foo: f}] } = obj; // f = 123
```

10.3.1 获取你需要的

解构一个对象，仅提取感兴趣的属性：

```
let { x: x } = { x: 7, y: 3 }; // x = 7
```

如果解构一个数组，可以选择仅提取一个前缀：

```
let [x,y] = ['a', 'b', 'c']; // x='a'; y='b';
```

10.4 模式是如何获取到内部的数据的？

在赋值中 `pattern = someValue` ，`pattern` 是如何获取 `someValue` 内部的数据的？

10.4.1 对象模式必须要数据是对象

在获取属性之前，对象模式将源解构成对象。这意味着解构对原始值也有效：

```
let {length : len} = 'abc'; // len = 3
let {toString: s} = 123; // s = Number.prototype.toString
```

10.4.1.1 对象解构过程中，值转换失败

转换成对象的过程不是通过 `Object()` 的，而是通过内部的 ToObject() 操作。`Object()` 从不失败：

```
> typeof Object('abc')
'object'
> var obj = {};
> Object(obj) === obj
true
> Object(undefined)
{}
> Object(null)
{}
```

如果转换目标是 `undefined` 或者 `null` ，则 `ToObject()` 会抛出一个 `TypeError` 异常。因此，下面的解构过程在获取属性之前就失败了：

```
let { prop: x } = undefined; // TypeError
let { prop: y } = null; // TypeError
```

因此，可以使用空的对象模式 `{}` 去检测一个值在解构过程中是否可转换成一个对象。就像上面看见的，仅 `undefined` 和 `null` 不能转换：

```
({} = [true, false]); // OK, Arrays are coercible to objects
({} = 'abc'); // OK, strings are coercible to objects

({} = undefined); // TypeError
({} = null); // TypeError
```

表达式两侧的括号是必要的，因为在 JavaScript 中语句不能以大括号开始。

10.4.2 数组模式通过可迭代工作

数组解构使用迭代器获取源的元素。因此，可以用数组的方式解构任何一个可迭代的值。让我们看一看可迭代的值的示例。

字符串是可迭代的：

```
let [x,...y] = 'abc'; // x='a'; y=['b', 'c']
```

别忘了，字符串的迭代器返回代码点（ code points ）（“ Unicode 字符”，21位），而不是代码单元（“ JavaScript 字符”，16位）。（更多关于 Unicode 的信息，参考“ Speaking JavaScript ”的“ Chapter 24. Unicode and JavaScript ”章）例如：

```
let [x,y,z] = 'a\uD83D\uDCA9c'; // x='a'; y='\uD83D\uDCA9'; z='c'
```

不能通过索引下标获取一个 Set 集合的元素，但是可以通过迭代器获取。因为，数组解构对 Set 有效：

```
let [x,y] = new Set(['a', 'b']); // x='a'; y='b';
```

`Set` 迭代器返回元素的顺序与元素插入的顺序一致，这就是为什么上述的解构结果总是一样的。

无穷序列。解构同样对无穷序列有效。生成器函数 `allNaturalNumbers()` 返回一个迭代器，生成0，1，2，...。

```
function* allNaturalNumbers() {
  for (let n = 0; ; n++) {
    yield n;
  }
}
```

下面的解构获取该无穷序列前面的三个元素。

```
let [x, y, z] = allNaturalNumbers(); // x=0; y=1; z=2
```

10.4.2.1 数组解构过程中，值转换失败

如果一个值有一个方法，该方法有一个键是 `Symbol.iterator` ，键值是一个对象，那么该值就是可迭代的。如果要解构的对象不可迭代，那么数组解构时就会抛出 `TypeError` 错误：

```
let x;
[x] = [true, false]; // OK, Arrays are iterable
[x] = 'abc'; // OK, strings are iterable
[x] = { * [Symbol.iterator]() { yield 1 } }; // OK, iterable

[x] = {}; // TypeError, empty objects are not iterable
[x] = undefined; // TypeError, not iterable
[x] = null; // TypeError, not iterable
```

在获取迭代元素之前 `TypeError` 异常就会抛出，这意味着可以使用空的数组模式 `[]` 来检查一个值是否可迭代：

```
[] = {}; // TypeError, empty objects are not iterable
[] = undefined; // TypeError, not iterable
[] = null; // TypeError, not iterable
```

10.5 如果有一部分没有匹配上

解构过程中如果要获取的目标在源上不存在，此时的处理方式与 JavaScript 处理不存在的属性和数组元素一样：内部相应部分匹配上 `undefined` 。如果该内部部分是一个变量，意味着这个变量会被设为 `undefined` ：

```
let [x] = []; // x = undefined
let {prop:y} = {}; // y = undefined
```

记住：如果去匹配 `undefined` ，对象模式和数组模式都会抛出 `TypeError` 异常。

10.5.1 默认值

默认值是模式的一个特性：如果某个部分（一个对象属性或者一个数组元素）在源里面没有匹配的内容，它会得到：

- 它的默认值（如果指定了默认值）
- undefined（否则）

也就是说，提供默认值是可选的。

让我们看一个例子。在下面的解构过程中，第0个元素在右侧没有匹配上。因此， `x` 对应上3，解构过程继续，最终导致 `x` 的值被设为3。

```
let [x=3, y] = []; // x = 3; y = undefined
```

也可以在对象模式中使用默认值：

```
let {foo: x=3, bar: y} = {}; // x = 3; y = undefined
```

10.5.1.1 `undefined` 使默认值生效

当某个部分没有匹配或者匹配上的是 `undefined` ，就会使用默认值：

```
let [x=1] = [undefined]; // x = 1
let {prop: y=2} = {prop: undefined}; // y = 2
```

此现象的原理将会在下一章阐述，在《默认参数》那一节。

10.5.1.2 默认值按需计算

默认值本身在需要的时候计算。换句话说，下面的解构：

```
let {prop: y=someFunc()} = someValue;
```

等同于：

```
let y;
if (someValue.prop === undefined) {
    y = someFunc();
} else {
    y = someValue.prop;
}
```

可以通过 `console.log()` 来观测到这个过程：

```
> function log(x) { console.log(x); return 'YES' }

> let [a=log('hello')] = [];
hello
> a
'YES'

> let [b=log('hello')] = [123];
> b
123
```

在第二个解构过程中，默认值没有被触发，所以没有调用 `log()` 。

10.5.1.3 默认值可以指向模式中其它变量

默认值可以指向任何变量，包括在同一个模式中的另一个变量：

```
let [x=3, y=x] = [];     // x=3; y=3
let [x=3, y=x] = [7];    // x=7; y=7
let [x=3, y=x] = [7, 2]; // x=7; y=2
```

但是，顺序很重要：变量 `x` 和 `y` 由左向右依次声明，如果在声明之前访问的话，就会产生一个 `ReferenceError` 异常：

```
let [x=y, y=3] = []; // ReferenceError
```

10.5.1.4 模式的默认值

到目前为止，我们值看到变量的默认值，也可以给模式设定默认值：

```
let [{ prop: x } = {}] = [];
```

这意味着什么？回忆一下默认值的规则：

> 如果模式中某部分在源中没有匹配，将会使用默认值[...]，解构过程继续。

上面的例子中，在索引0处的元素没有匹配，解构过程进入如下步骤：

```
let { prop: x } = {}; // x = undefined
```

你可以把模式 `{ prop: x }` 替换成变量 `pattern` 来更容易地看出为什么会这样：

```
let [pattern = {}] = [];
```

更复杂的默认值。让我们更深入地探索模式的默认值。在下面的例子中，我们通过默认值 `{ prop: 123 }` 给 `x` 赋上一个值：

```
let [{ prop: x } = { prop: 123 }] = [];
```

因为在索引为0处的数组元素在右侧没有相应匹配，解构过程以如下所示的形式继续，并且 `x` 的值被设为123。

```
let { prop: x } = { prop: 123 };  // x = 123
```

然而，如果右侧在索引0处有一个元素， `x` 将不会以这种方式被赋上一个值，因为并没有触发默认值。

```
let [{ prop: x } = { prop: 123 }] = [{}];
```

这种情形下，解构过程像下面这样继续：

```
let { prop: x } = {}; // x = undefined
```

因此，如果想在对象或者属性不存在的时候，让 `x` 被赋上值123，需要为 `x` 自身指定上默认值：

```
let [{ prop: x=123 } = {}] = [{}];
```

在这里，解构过程像下面这样继续，不管右侧是 `[{}]` ，还是 `[]` 。

```
let { prop: x=123 } = {}; // x = 123
```

> 仍然感到迷惑
> 后面有个章节从算法角度解释了解构过程，这将会给你一个另外的视角。

10.6 更多对象解构特性

10.6.1 属性值缩写

属性值缩写是对象字面量的特性：如果属性的值是通过一个变量提供的，这个变量的名字和键名一样，就可以省略这个键名。这对结构也同样有效：

```
let { x, y } = { x: 11, y: 8 }; // x = 11; y = 8
```

该声明等同于：

```
let { x: x, y: y } = { x: 11, y: 8 };
```

10.6.2 计算属性键

计算属性键是另一个对象字面量特性，同样对解构有效：可以通过一个表达式指定属性键，把表达式放在中括号里面：

```
const FOO = 'foo';
let { [FOO]: f } = { foo: 123 }; // f = 123
```

计算属性值允许你解构键是 symbol 的属性：

```
// Create and destructure a property whose key is a symbol
const KEY = Symbol();
let obj = { [KEY]: 'abc' };
let { [KEY]: x } = obj; // x = 'abc'

// Extract Array.prototype[Symbol.iterator]
let { [Symbol.iterator]: func } = [];
console.log(typeof func); // function
```

10.7 更多数组解构特性

10.7.1 省略

在解构过程中，省略使用数组“空洞”语法来跳过元素：

```
let [,,x] = ['a', 'b', 'c', 'd']; // x = 'c'
```

10.7.2 剩余操作符（...）

剩余操作符使你能够提取剩余的数组元素到一个数组里面。你可以在数组模式的最后的部分使用这个操作符：

```
let [x, ...y] = ['a', 'b', 'c']; // x='a'; y=['b', 'c']
```

> 剩余操作符提取数据，数组字面量和函数调用都可以使用相同的扩展操作符语法（...），下一章会讲解这种用法。

如果该操作符没找到任何元素，将会使操作数匹配上一个空数组。也就是说，剩余操作符从不产生 `undefined` 或者 `null` 。例如：

```
let [x, y, ...z] = ['a']; // x='a'; y=undefined; z=[]
```

剩余操作符的操作数不一定是一个变量，也可以是模式：

```
let [x, ...[y, z]] = ['a', 'b', 'c'];
    // x = 'a'; y = 'b'; z = 'c'
```

剩余操作符触发了如下解构过程：

```
[y, z] = ['b', 'c']
```

> 扩展操作符（...）看起来和剩余操作符一模一样，但是它用于函数调用和数组字面量（不是在解构模式里面）。

不仅仅能赋值给变量

如果通过解构赋值，每一个赋值对象都可以是任何的通常能在左侧接受赋值的东西，包括指向属性的引用（ `obj.prop` ）和指向数组元素的引用（ `arr[0]` ）。

```
let obj = {};
let arr = [];

({ foo: obj.prop, bar: arr[0] }) = { foo: 123, bar: true };

console.log(obj); // {prop:123}
console.log(arr); // [true]
```

也可以通过剩余操作符（...）赋值给对象属性和数组元素：

```
let obj = {};
[first, ...obj.rest] = ['a', 'b', 'c'];
    // first = 'a'; obj.rest = ['b', 'c']
```

10.9 解构陷阱

在使用解构的时候，要考虑两件事情：

- 一条语句不能以大括号开始。
- 在解构过程中，要么声明变量，要么赋值给它们，但是两件事不能一起做。

下面的两节讲述了详细内容。

10.9.1 一条语句不要以大括号开始

因为代码块以大括号开始，因此语句一定不要以大括号开始。这在对象解构赋值的时候会失败：

```
{ a, b } = someObject; // SyntaxError
```

变通的方法是把整个表达式放到小括号中：

```
({ a, b } = someObject); // ok
```

10.9.2 对于已经存在的变量，不能同时声明和赋值

在一个解构变量声明的过程中，左侧每一个变量都声明了。在下面的例子中，我们尝试声明变量 `b` ，并且使用已经存在的变量 `f` ，这种做法将会失败：

```
let f;
...
let { foo: f, bar: b } = someObject;
    // During parsing (before running the code):
    // SyntaxError: Duplicate declaration, f
```

修复的方法是事先声明好 `b` ，然后解构赋值：

```
let f;
...
let b;
({ foo: f, bar: b }) = someObject;
```

10.10 解构示例

让我们以几个更小的例子开始。

`for-of` 循环支持解构：

```
let map = new Map().set(false, 'no').set(true, 'yes');
for (let [key, value] of map) {
  console.log(key + ' is ' + value);
}
```

可以使用解构来交换变量值，引擎会优化这个过程，因此不会创建额外的数组。

```
[a, b] = [b, a];
```

可以使用解构来分离数组：

```
let [first, ...rest] = ['a', 'b', 'c'];
    // first = 'a'; rest = ['b', 'c']
```

10.10.1 解构返回值

一些内置的 JavaScript 操作返回数组，解构可以辅助处理返回的数组：

```
let [all, year, month, day] =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
```

10.10.2 多个返回值

为了看一下很有用的多值返回，让我们事先一个函数 `findElement(a, p)` ，该函数查找数组 `a` 中第一个使函数 `p` 返回 `true` 的元素。问题是：这个函数要返回什么？有时可能对满足条件的那个元素感兴趣，有时是元素的索引，有时是两者都感兴趣。下面的实现两者都返回。

```
function findElement(array, predicate) {
    for (let [index, element] of array.entries()) { // (A)
        if (predicate(element)) {
            return { element, index }; // (B)
        }
    }
    return { element: undefined, index: -1 };
}
```

在 A 行，数组方法 `entries()` 返回一个迭代器，迭代器的每个元素是一个 `[index, element]` 对。每次迭代解构一个元素。在 B 行，使用属性值缩写返回对象 `{ element: element, index: index }` 。

使用 `findElement()` 。在下面的例子中，一些 ECMAScript 6 特性允许我们书写更加简明的代码：回调函数是一个箭头函数，返回值通过对象模式的属性值简写来解构。

```
let arr = [7, 8, 6];
let {element, index} = findElement(arr, x => x % 2 === 0);
    // element = 8, index = 1
```

由于 `index` 和 `element` 也指向属性键，它们书写的顺序是无关紧要的：

```
let {index, element} = findElement(···);
```

我们成功处理了同事使用索引和元素的情况。如果只对其中某一个感兴趣怎么办？事实证明，感谢 ECMAScript 6 ，我们的实现也同样能处理这种情形，并且相对于返回单一值的函数，语法开销是最小的。

```
let a = [7, 8, 6];

let {element} = findElement(a, x => x % 2 === 0);
    // element = 8

let {index} = findElement(a, x => x % 2 === 0);
    // index = 1
```

每一次，仅提取需要的属性值。

10.11 解构算法

本节从不同的角度来看解构过程：一个递归的模式匹配算法。

> 这个不同的角度应该对理解默认值很有帮助。如果你感觉自己没有完全理解解构过程，那么继续读下去。

在最后，我将会使用这个算法解释下面两个函数声明的不同之处。

```
function move({x=0, y=0} = {})         { ··· }
function move({x, y} = { x: 0, y: 0 }) { ··· }
```

10.11.1 算法

解构赋值看起来像这样：

```
«pattern» = «value»
```

我们想使用 `pattern` 从 `value` 里面获取数据。我现在将会描述一种算法来做这件事，这在函数式编程中被称为模式匹配（ pattern matching ）（简称：匹配）。该算法指定操作符 `←` （“匹配”）用于表示解构赋值中给 `pattern` 匹配上一个 `value` 并赋值给变量：

```
«pattern» ← «value»
```

该算法通过递归的规则执行，这些递归规则在←操作符两侧的操作数都会发生。这个声明性的符号可能不好适应，但是它让该算法的说明更加简洁明了。每一个规则都有两部分：

- 头部指明该规则应用到的操作数。
- 主体部分指明接下来做什么。

我仅展示解构赋值的算法。解构变量声明和解构参数定义是类似的。

我也不会讲解高级的特性（计算属性键；属性值缩写；对象属性和数组元素作为赋值目标），仅仅讲解基础的东西。

10.11.1.1 模式

一个模式是下列的几种形式之一：

- 一个变量： `x`
- 一个对象模式： `{«properties»}`

- 一个数组模式：`[«elements»]`

后面每节描述了这三种情况中的一种。

10.11.1.2 变量

- （1）`x ← value`（包括 `undefined` 和 `null`）

```
x = value
```

10.11.1.3 对象模式

- （2a）`{«properties»} ← undefined`

```
throw new TypeError();
```

- （2b）`{«properties»} ← null`

```
throw new TypeError();
```

- （2c）`{key: «pattern», «properties»} ← obj`

```
«pattern» ← obj.key
{«properties»} ← obj
```

- （2d）`{key: «pattern» = default_value, «properties»} ← obj`

```
let tmp = obj.key;
if (tmp !== undefined) {
  «pattern» ← tmp
} else {
  «pattern» ← default_value
}
{«properties»} ← obj
```

- （2e）`{} ← obj`

```
// No properties left, nothing to do
```

10.11.1.4 数组模式

数组模式和迭代器。数组解构算法以数组模式和一个迭代器开始：

- （3a） `[«elements»] ← non_iterable` `assert(!isIterable(non_iterable))`

```
throw new TypeError();
```

- （3b） `[«elements»] ← iterable` `assert(isIterable(iterable))`

```
let iterator = iterable[Symbol.iterator]();
«elements» ← iterator
```

辅助函数：

```
function isIterable(value) {
    return (value !== null
        && typeof value === 'object'
        && typeof value[Symbol.iterator] === 'function');
}
```

数组元素和迭代器。算法以模式和迭代器的元素继续（从迭代器中取值）。

- （3c） `«pattern», «elements» ← iterator`

```
«pattern» ← getNext(iterator) // undefined after last item
«elements» ← iterator
```

- （3d） `«pattern» = default_value, «elements» ← iterator`

```
let tmp = getNext(iterator);  // undefined after last item
if (tmp !== undefined) {
  «pattern» ← tmp
} else {
  «pattern» ← default_value
}
«elements» ← iterator
```

- （3e） `, «elements» ← iterator (hole, elision)`

```
getNext(iterator); // skip
«elements» ← iterator
```

- （3f） `...«pattern» ← iterator` （总是最后一部分！）

```
let tmp = [];
for (let elem of iterator) {
  tmp.push(elem);
}
«pattern» ← tmp
```

- （3g） `← iterator`

辅助函数：

```
function getNext(iterator) {
    let {done,value} = iterator.next();
    return (done ? undefined : value);
}
```

10.11.2 应用算法

下面的函数定义有命名的参数，一种有时称作可选对象（options object）的技术，在 参数处理章节 有解释。参数使用了解构和默认值，这样 `x` 和 `y` 就可以省略不传了。但是对象参数也可以不传，就像下面代码中最后一行一样。此特性通过在函数声明头部的 `={}` 起作用。

```
function move1({x=0, y=0} = {}) {
    return [x, y];
}
move1({x: 3, y: 8}); // [3, 8]
move1({x: 3}); // [3, 0]
move1({}); // [0, 0]
move1(); // [0, 0]
```

但是为什么要像上述代码片段一样定义参数呢？为什么不是像下面这样的呢 - 这也是完全合法的 ES6 代码？

```
function move2({x, y} = { x: 0, y: 0 }) {
    return [x, y];
}
```

为了验证为什么 `move1()` 是正确的，让我们在两个例子中使用这两种函数。在做这件事之前，让我们看看传递的参数是如何匹配解析的。

10.11.2.1 背景：通过匹配传递参数

对于函数调用，形参（在函数定义里面）匹配实参（在函数调用里面）。举个例子，使用下面的函数定义和下面的函数调用：

```
function func(a=0, b=0) { ··· }
func(1, 2);
```

参数 `a` 和 `b` 和下面的解构过程类似地被设置。

```
[a=0, b=0] ← [1, 2]
```

10.11.2.2 使用 `move2()`

让我们看一下 `move2()` 的解构过程是怎样的。

示例**1**。 `move2()` 会导致该解构过程：

```
[{x, y} = { x: 0, y: 0 }] ← []
```

左侧仅有的数组元素在右侧并没有匹配的内容，这就是为什么 `{x, y}` 匹配上了默认值，而不是来自于右侧的数据（规则 3b ， 3d ）：

```
{x, y} ← { x: 0, y: 0 }
```

该解构过程导致下面的两个赋值（规则 2c ，1）：

```
x = 0;
y = 0;
```

这就是仅有的使用默认值的情况。

示例**2**。让我们看看函数调用 `move2({z:3})` ，这会导致如下的解构过程：

```
[{x, y} = { x: 0, y: 0 }] ← [{z:3}]
```

在右侧数组的索引0处有一个数组元素。因此，默认值会被忽略，下一步是（规则 3d ）：

```
{x, y} ← { z: 3 }
```

这会导致 `x` 和 `y` 都被设置为 `undefined` ，这不是我们想要的。

10.11.2.3 使用 `move1()`

让我们尝试 `move1()` 。

示例**1**： `move1()`

```
[{x=0, y=0} = {}] ← []
```

在右侧数组的索引0处没有一个数组元素，所以使用默认值（规则 3d ）：

```
{x=0, y=0} ← {}
```

左侧包含属性值缩写，解构过程相当于：

```
{x: x=0, y: y=0} ← {}
```

属性 `x` 和属性 `y` 在右侧都没有匹配的内容。因此，使用默认值，下一步是如下所示的解构过程（规则 2d ）：

```
x ← 0
y ← 0
```

这会导致如下的赋值（规则1）：

```
x = 0
y = 0
```

示例**2**： `move1({z:3})`

```
[{x=0, y=0} = {}] ← [{z:3}]
```

就像示例1一样，属性 `x` 和 `y` 在右侧没有匹配的内容，所以使用默认值：

```
x = 0
y = 0
```

10.11.3 结论

这些例子展示了默认值是模式部分（对象属性或者数组元素）的特性。如果某个部分没有匹配上，或者匹配到了 `undefined` ，那么默认值就生效了。也就是说，模式匹配上了默认值。

11 参数处理

在 ECMAScript 6 中，参数处理得到了显著地升级。现在支持默认参数值，剩余参数（ rest parameters ）（可变参数）和解构。

> 在本章，对解构过程熟悉会很有帮助（该过程在上一章讲解了）。

11.1 概览

默认参数值：

```
function findClosestShape(x=0, y=0) {
    // ...
}
```

剩余参数：

```
function format(pattern, ...params) {
    return params;
}
console.log(format('a', 'b', 'c')); // ['b', 'c']
```

运用解构的命名参数：

```
function selectEntries({ start=0, end=-1, step=1 } = {}) {
    // The object pattern is an abbreviation of:
    // { start: start=0, end: end=-1, step: step=1 }

    // Use the variables `start`, `end` and `step` here
    ...
}

selectEntries({ start: 10, end: 30, step: 2 });
selectEntries({ step: 3 });
selectEntries({});
selectEntries();
```

11.1.1 扩展操作符（ **Spread operator** ）（...）

在函数和构造器的调用中，扩展操作符把可迭代的值转换成参数：

```
> Math.max(-1, 5, 11, 3)
11
> Math.max(...[-1, 5, 11, 3])
11
> Math.max(-1, ...[-1, 5, 11], 3)
11
```

在数组字面量中，扩展操作符将可迭代的一组值转换成数组元素：

```
> [1, ...[2,3], 4]
[1, 2, 3, 4]
```

11.2 参数解构

在 ES6 中，处理参数和通过形参解构实参的过程是一样的。也就是说，下面的函数调用：

```
function func(«FORMAL_PARAMETERS») {
    «CODE»
}
func(«ACTUAL_PARAMETERS»);
```

大致就是：

```
{
    let [«FORMAL_PARAMETERS»] = [«ACTUAL_PARAMETERS»];
    {
        «CODE»
    }
}
```

例子 - 下面的函数调用：

```
function logSum(x=0, y=0) {
    console.log(x + y);
}
logSum(7, 8);
```

变成：

```
{
    let [x=0, y=0] = [7, 8];
    {
        console.log(x + y);
    }
}
```

11.3 默认参数值

ECMAScript 6 允许给参数指定默认值：

```
function f(x, y=0) {
  return [x, y];
}
```

省略第二个参数就会触发默认值：

```
> f(1)
[1, 0]
> f()
[undefined, 0]
```

当心 - `undefined` 也会触发默认值：

```
> f(undefined, undefined)
[undefined, 0]
```

默认值仅在需要的时候在后台计算：

```
> const log = console.log.bind(console);
> function g(x=log('x'), y=log('y')) {return 'DONE'}
> g()
x
y
'DONE'
> g(1)
y
'DONE'
> g(1, 2)
'DONE'
```

11.3.1 为什么 `undefined` 会触发默认值？

为什么 `undefined` 应该被当成一个缺失的参数，或者对象或数组的缺失部分？这个问题不是显而易见的。这么做的原理是它使你能够把默认值的定义转移到其它地方去。让我们看两个例子。

第一个例子（来源： Rick Waldron's TC39 meeting notes from 2012-07-24 ），在 `setOptions()` 中不需要定义一个默认值，可以把这个任务转移到 `setLevel()` 。

```
function setLevel(newLevel = 0) {
    light.intensity = newLevel;
}
function setOptions(options) {
    // Missing prop returns undefined => use default
    setLevel(options.dimmerLevel);
    setMotorSpeed(options.speed);
    ...
}
setOptions({speed:5});
```

第二个例子，`square()` 不必为 `x` 定义一个默认值，可以将这个任务转移到 `multiply()` ：

```
function multiply(x=1, y=1) {
    return x * y;
}
function square(x) {
    return multiply(x, x);
}
```

默认值更进一步强调了 `undefined` 指代不存在的内容而 `null` 指代空（emptiness）的语意。

11.3.2 在默认值中使用其它变量

在参数的默认值中，可以使用任何变量，包括其它参数：

```
function foo(x=3, y=x) { ··· }
foo();     // x=3; y=3
foo(7);    // x=7; y=7
foo(7, 2); // x=7; y=2
```

然后，顺序很重要：参数从左到右声明，因此在默认值中，如果去访问一个还没声明的参数，将会抛出 `ReferenceError` 异常。

默认值存在于它们自己的作用域中，该作用域介于包裹函数的“外部”作用域和函数体“内部”作用域之间。因此，在默认值中不能访问函数内部的变量：

```
let x = 'outer';
function foo(a = x) {
    let x = 'inner';
    console.log(a); // outer
}
```

在上面的例子中，如果没有外部的 `x` ，默认值 `x` 将会产生一个 `ReferenceError` 异常。

如果默认值是闭包，该限制可能让人非常吃惊：

```
function foo(callback = () => BAR) {
    const BAR = 3; // can’t be accessed from default value
    callback();
}
foo(); // ReferenceError
```

11.4 剩余参数（ Rest parameters ）

将剩余操作符（...）放在最后一个形参的前面意味着该形参会接收到所有剩下的实参（这些实参放置在一个数组中，然后传给这个形参）。

```
function f(x, ...y) {
    ...
}
f('a', 'b', 'c'); // x = 'a'; y = ['b', 'c']
```

如果没有剩余的参数，剩余参数将会被设置为空数组：

```
f(); // x = undefined; y = []
```

> 扩展操作符（...）看起来和剩余操作符一模一样，但是它用于函数调用和数组字面量（而不是在解构模式里面）。

11.4.1 禁止再使用 `arguments` ！

剩余参数可以完全替代 JavaScript 的臭名昭著的特殊变量 `arguments` 。剩余参数的优点是它总是数组：

```
// ECMAScript 5: arguments
function logAllArguments() {
    for (var i=0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}

// ECMAScript 6: rest parameter
function logAllArguments(...args) {
    for (let arg of args) {
        console.log(arg);
    }
}
```

11.4.1.1 混合解构过程和访问解构值

一个有趣的 `arguments` 的特性就是定义普通参数的同时，也会拿到所有参数的一个数组：

```
function foo(x=0, y=0) {
    console.log('Arity: '+arguments.length);
    ...
}
```

在这种情形下，如果将一个剩余参数和数组解构结合起来，就可以避免使用 `arguments` 了。最终的代码相对较长，但是更加清晰：

```
function foo(...args) {
    let [x=0, y=0] = args;
    console.log('Arity: '+args.length);
    ...
}
```

这对命名参数也有效（可选对象）：

```
function bar(options = {}) {
    let { namedParam1, namedParam2 } = options;
    ...
    if ('extra' in options) {
        ...
    }
}
```

11.4.1.2 `arguments` 是可迭代的

在 ECMAScript 6 中， `arguments` 是可迭代的，这意味着你可以使用 `for-of` 循环和扩展操作符：

```
> (function () { return typeof arguments[Symbol.iterator] }())
'function'
> (function () { return Array.isArray([...arguments]) }())
true
```

11.5 模拟命名参数

在编程语言中调用一个函数（或者方法）的时候，你必须将实参（调用者指定的）映射到形参（函数中定义的）。有两种方式来完成这种映射：

- 位置相关的参数通过位置来映射。第一个实参映射到第一个形参，第二个实参映射到第二个形参，以此类推。
- 命名参数使用名字（标签）来实现这种映射。名字和在函数定义中的形参关联起来，在函数调用中给实参打上标记。命名参数以何种顺序出现并不重要，因为它们被正确标记了。

命名参数主要有两个好处：在函数调用中它们提供了参数描述，同时对于可选对象也能轻松应对。我将会首先介绍这些好处，然后向你展示在 JavaScript 中如何通过对象字面量模拟命名参数。

11.5.1 作为描述的命名参数

只要一个函数有超过一个的参数，你可能就会对每一个参数用来做什么感到迷惑。例如，假设有一个函数 `selectEntries()` ，返回一组来自于数据库的条目。给出函数调用：

```
selectEntries(3, 20, 2);
```

这两个数字是什么意思？ Python 支持命名参数，所以在 Python 中可以很轻松地说明发生了什么：

```
# Python syntax
selectEntries(start=3, end=20, step=2)
```

11.5.2 可选的命名参数

可选的位置相关的参数，仅在位于形参末尾的时候被省略才会正常工作。其它情形，必须插入占位符（比如 `null` ），以便于剩下的参数能正确对上位置。

对于可选的命名参数来说，这不是问题。你可以轻易地省略任何一个参数。下面有一些例子：

```
# Python syntax
selectEntries(step=2)
selectEntries(end=20, start=3)
selectEntries()
```

11.5.3 在 JavaScript 中模拟命名参数

对于命名参数，JavaScript 并没有像 python 和其它很多语言一样本地支持。但是有一种合理的优雅的模拟方式：通过对象字面量的命名参数，以单个实参的形式传入。当你使用这种方式的时候，一次 `selectEntries()` 的调用看起来像这样：

```
selectEntries({ start: 3, end: 20, step: 2 });
```

函数接收到一个对象，该对象有 `start` ， `end` 和 `step` 属性。你可以省略任何一个：

```
selectEntries({ step: 2 });
selectEntries({ end: 20, start: 3 });
selectEntries();
```

在 ECMAScript 5 中，你可能像下面这样实现 `selectEntries()` ：

```
function selectEntries(options) {
    options = options || {};
    var start = options.start || 0;
    var end = options.end || getDbLength();
    var step = options.step || 1;
    ...
}
```

在 ECMAScript 6 中，你可以使用解构，看起来像这样：

```
function selectEntries({ start=0, end=-1, step=1 }) {
    ...
}
```

如果你调用 `selectEntries()` 不传入参数，解构就会失败，因为你不可能让一个对象模式匹配上 `undefined` 。这个问题可以通过默认值来修复。在下面的代码中，如果没有传入任何参数的话，对象模式会匹配上 `{}` 。

```
function selectEntries({ start=0, end=-1, step=1 } = {}) {
    ...
}
```

你也可以将位置相关的参数和命名参数结合起来。通常的做法是把命名参数放在最后：

```
someFunc(posArg1, { namedArg1: 7, namedArg2: true });
```

原则上， JavaScript 引擎会优化这种模式，这样就不会创建中间对象，因为调用方的对象字面量和函数定义处的对象模式都是静态的。

在 JavaScript 中，此处展示的命名参数形式有时被称为可选项或者可选对象（比如， jQuery 文档）。

11.6 参数解构示例

11.6.1 提示：箭头函数中单个参数两边的括号

在后面的内容中，我会偶尔使用箭头函数。因此，一个快速的提示：如果一个箭头函数只有一个参数，并且那个参数是一个标识符，你可以省略参数两边的括号。例如，在下面的 REPL 交互中， `x` 两边没有括号：

```
> [1,2,3].map(x => 2 * x)
[ 2, 4, 6 ]
```

然而，当单个参数不是一个标识符的时候必须带上括号：

```
> [[1,2], [3,4]].map(([a,b]) => a + b)
[ 3, 7 ]

> [1, undefined, 3].map((x='yes') => x)
[ 1, 'yes', 3 ]
```

更多详细内容在箭头函数那一章有讲解。

11.6.2 forEach() 和解构

在 ECMAScript 6 中你可能将会大量使用 `for-of` 循环，但是数组方法 `forEach()` 同样从解构中获益。更精确地说，它的回调函数获益了。

第一个例子：在数组中解构数组。

```
let items = [ ['foo', 3], ['bar', 9] ];
items.forEach(([word, count]) => {
    console.log(word+' '+count);
});
```

第二个例子：解构数组中的对象。

```
let items = [
    { word:'foo', count:3 },
    { word:'bar', count:9 },
];
items.forEach(({word, count}) => {
    console.log(word+' '+count);
});
```

11.6.3 转换 Map

一个 ECMAScript 6 Map 没有 `map()` 方法（像数组一样的）。因此，必须要这样：

- 1、转换成一个 `[key,value]` 对的数组。
- 2、在数组上调用 `map()` 方法。
- 3、将结果转换回 Map 。

这个过程看起来像下面这样。

```
let map0 = new Map([
    [1, 'a'],
    [2, 'b'],
    [3, 'c'],
]);

let map1 = new Map( // step 3
    [...map0] // step 1
    .map(([k, v]) => [k*2, '_'+v]) // step 2
);
// Resulting Map: {2 -> '_a', 4 -> '_b', 6 -> '_c'}
```

11.6.4 处理通过 Promise 返回的数组

工具方法 `Promise.all()` 以如下方式运作：

- 输入：一组 Promises 。
- 输出：在最后输入的 Promise 返回了的时候，就会有一个 Promise 返回一个数组，这个数组包含了输入的 Promises 返回的结果。

解构可以帮助处理 `Promise.all()` 最终返回的数组：

```
let urls = [
    'http://example.com/foo.html',
    'http://example.com/bar.html',
    'http://example.com/baz.html',
];

Promise.all(urls.map(downloadUrl))
.then(([fooStr, barStr, bazStr]) => {
    ...
});

// This function returns a Promise that resolves to
// a string (the text)
function downloadUrl(url) {
    return fetch(url).then(request => request.text());
}
```

`fetch()` 就是一个基于 Promise 的 `XMLHttpRequest` 。这是 Fetch 标准的一部分。

11.7 编码风格小建议

本节会提到几个描述参数定义的技巧。很高明，但是也有缺陷：使代码看起来很乱，并且更难理解。

11.7.1 可选参数

我偶尔使用参数默认值 `undefined` 来标记某个参数是可选的（除非这个参数已经有一个默认值了）：

```
function foo(requiredParam, optionalParam = undefined) {
    ...
}
```

11.7.2 必需的参数

在 ECMAScript 5 中，有几种可选方法用于确保提供了必需的参数，但是都显得相当笨拙：

```
function foo(mustBeProvided) {
    if (arguments.length < 1) {
        throw new Error();
    }
    if (! (0 in arguments)) {
        throw new Error();
    }
    if (mustBeProvided === undefined) {
        throw new Error();
    }
    ...
}
```

在 ECMAScript 6 中，你可以（大量）使用默认值来写出更加简洁明了的代码（来自于 Allen Wirfs-Brock 的想法）：

```
/**
 * Called if a parameter is missing and
 * the default value is evaluated.
 */
function mandatory() {
    throw new Error('Missing parameter');
}
function foo(mustBeProvided = mandatory()) {
    return mustBeProvided;
}
```

交互模式：

```
> foo()
Error: Missing parameter
> foo(123)
123
```

11.7.3 指定最多参数个数

本节展示三种方法来指定最多参数个数。例子中的函数 `f` 参数个数不超过2 - 如果调用者传入的参数超过2个，会抛出一个错误。

第一个方法是在剩余参数 `args` 中搜集所有的实参，然后检查它的长度。

```
function f(...args) {
    if (args.length > 2) {
        throw new Error();
    }
    // Extract the real parameters
    let [x, y] = args;
}
```

第二个方法依赖于不想要的参数出现在了剩余参数 `empty` 中。

```
function f(x, y, ...empty) {
    if (empty.length > 0) {
        throw new Error();
    }
}
```

第三种方法使用一个守卫值，如果传入了第三个参数，就得不到守卫值了。警告一点就是如果传入的第三个参数的值是 `undefined` ，也会触发默认值 `OK` 。

```
const OK = Symbol();
function f(x, y, arity=OK) {
    if (arity !== OK) {
        throw new Error();
    }
}
```

悲剧的是，每一个方法写出的代码都看起来乱糟糟的，并且概念混乱。我倾向于检查 `arguments.length` ，但是又希望废弃 `arguments` 。

```
function f(x, y) {
    if (arguments.length > 2) {
        throw new Error();
    }
}
```

11.8 扩展运算符（...）

扩展操作符（...）和剩余操作符看起来很像，但是两者是对立的：

- 剩余操作符提取数组，用于剩余参数和解构。
- 扩展操作符将数组元素转换成函数调用的参数或者数组字面量的元素。

11.8.1 函数和方法调用中的扩展操作

`Math.max()` 是一个很好的例子，用于说明扩展操作符在方法调用中是如何运作的。 `Math.max(x1, x2, ···)` 返回值最大的参数。该方法接收任意数目的参数，但是不接收数组。扩展操作符弥补了这个缺陷：

```
> Math.max(-1, 5, 11, 3)
11
> Math.max(...[-1, 5, 11, 3])
11
```

相对于剩余操作符，你可以在参数序列的任何部分使用扩展操作符：

```
> Math.max(-1, ...[-1, 5, 11], 3)
11
```

另一个例子是 JavaScript 没有一种方式来追加一个数组的元素到另一个数组的后面。但是，数组确实有 `push(x1, x2, ...)` 方法，该方法将所有传入的参数追加到接收者。下面的代码展示了可以如何使用 `push()` 追加 `arr2` 中的元素到 `arr1` 中去。

```
let arr1 = ['a', 'b'];
let arr2 = ['c', 'd'];

arr1.push(...arr2);
// arr1 is now ['a', 'b', 'c', 'd']
```

11.8.2 构造器中的扩展操作

除了函数和方法调用之外，扩展操作符也对构造器调用生效：

```
new Date(...[1912, 11, 24]) // Christmas Eve 1912
```

这在 ECMAScript 5 中是难以实现的。

11.8.3 数组中的扩展操作

扩展操作符也可用于数组：

```
> [1, ...[2,3], 4]
[1, 2, 3, 4]
```

这让连接数组变得很方便：

```
let x = ['a', 'b'];
let y = ['c'];
let z = ['d', 'e'];

let arr = [...x, ...y, ...z]; // ['a', 'b', 'c', 'd', 'e']
```

11.8.3.1 将可迭代的或者类数组的对象转换成数组

扩展操作符能够将任何可迭代的对象转换成数组：

```
let arr = [...someIterableObject];
```

将 Set 转换成数组：

```
let set = new Set([11, -1, 6]);
let arr = [...set]; // [11, -1, 6]
```

自定义的可迭代对象也可以用同样的方式转换成数组：

```
let obj = {
    * [Symbol.iterator]() {
        yield 'a';
        yield 'b';
        yield 'c';
    }
};
let arr = [...obj]; // ['a', 'b', 'c']
```

注意，就像 `for-of` 循环，扩展操作符仅对可迭代对象有效。大多数重要的对象都是可迭代的：数组，Map，Sets 和 `arguments` 。大多数 DOM 数据结构最终也将会变得可迭代。

你应该遇到过一些对象不可迭代，但是是类数组的（索引化的元素加上一个 `length` 属性），你可以使用 `Array.from()` 将它转换成一个数组：

```
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};

// ECMAScript 5:
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']

// ECMAScript 6:
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']

// TypeError: Cannot spread non-iterable object.
let arr3 = [...arrayLike];
```

12 ECMAScript 6 中的可调用实体

本章给出在 ES6 中关于如何正确使用可调用实体（通过函数调用，方法调用等等）的建议，包含三节：

- 可调用实体的概览。
- 适合何种可调用实体的建议。
- 两种方法调用的检测，在 ES6 中是如何改变的：
    - 发送方法调用消息，例如 `obj.m(x, y)`
    - 直接方法调用，例如 `obj.m.call(obj, x, y)`

12.1 ECMAScript 6 中的可调用实体

在 ES6 中，有如下可调用的实体：

- 传统的函数（通过函数表达式和函数声明创建）
- 生成器函数（通过生成器函数表达式和生成器函数声明创建）
- 箭头函数（只有一种表达式的形式）
- 方法（通过对象字面量和类定义中的方法定义创建）
- 生成器方法（通过对象字面量和类类定义的生成器方法定义创建）
- 类（通过类表达式和类声明创建）

注意我区分了：

- 这种实体：例如传统函数
- 创建实体的语法：例如函数表达式和函数声明

尽管它们的表现差异很大（后面讲解），但是所有实体都是函数。例如：

```
> typeof (() => {}) // arrow function
'function'
> typeof function* () {} // generator function
'function'
> typeof class {} // class
'function'
```

接下来，我们详细学习每一种可调用实体。

12.1.1 ES6 中的调用方式

某些调用可以发生在任何地方，其它调用被限制在指定的区域。

12.1.1.1 可以在任何地方发生的调用

在 ES6 中有三种调用可以发生在任何地方：

- 函数调用： `func(3, 1)`
- 方法调用： `obj.method('abc')`
- 构造器调用： `new Constr(8)`

对于函数调用，记住这点很重要：大多数 ES6 代码会被包含进模块，并且模块体隐式地设为了严格模式。

12.1.1.2 通过 `super` 调用的方式被限制在指定的区域

两种调用可以通过 `super` 关键字发生；它们的使用被限制在指定的区域：

- 父方法调用： `super.method('abc')`

  仅在对象字面量或者继承类定义的方法定义中可用。
- 父构造器调用： `super(8)`

  仅在继承类特殊的方法 `constructor()` 中可用。

12.1.1.3 非方法函数（ non-method function ）和方法

非方法函数和方法之间的区别在 ES6 中变得更加明显了。现在两者都有特殊的实体，并且只有它们能做相应的事情：

- 箭头函数就是非方法函数。它从父作用域中拉取 `this` 和其它变量（“词法的 `this` ”）。
- 方法定义就是方法。它支持 `super` ，指向父属性和实现父方法调用。

12.1.2 传统的函数

有些函数来自于 ES5 。有两种方式创建这些函数：

- 函数表达式：

```
const foo = function (x) { ··· };
```

- 函数声明：

```
function foo(x) { ··· }
```

`this` 规则：

- 函数调用：在严格模式下， `this` 为 `undefined` ，宽松模式下为全局对象。
- 方法调用： `this` 是方法调用消息的接收者（或者是 `call/apply` 的第一个参数）。
- 构造器调用： `this` 是新创建的实例。

12.1.3 生成器函数

生成器函数在生成器章节介绍。它们的语法和传统的函数类似，但是有一个额外的星号：

- 生成器函数表达式：

```
const foo = function* (x) { ··· };
```

- 函数声明：

```
function* foo(x) { ··· }
```

`this` 规则如下所示。注意 `this` 绝不会指向当前生成器对象。

- 函数/方法调用：和传统的函数一样处理 `this` 。该调用的返回值是生成器对象。
- 构造器调用：在一个生成器函数中访问 `this` 会引起 `ReferenceError` 。构造器调用的结果是一个生成器对象。

12.1.4 方法定义

方法定义在对象字面量中出现：

```
let obj = {
    add(x, y) {
        return x + y;
    }, // comma is required
    sub(x, y) {
        return x - y;
    }, // comma is optional
};
```

在类定义中：

```
class AddSub {
    add(x, y) {
        return x + y;
    } // no comma
    sub(x, y) {
        return x - y;
    } // no comma
}
```

方法定义是唯一一个可以使用 `super` 指向父属性的地方。Only method definitions that use super produce functions that have the property [[HomeObject]], which is required for that feature (details are explained in the chapter on classes).

规则：

- 函数调用：如果取出一个方法，像函数一样调用它，这个方法就表现得和传统函数一样了。
- 方法调用：和传统函数一样，但是额外地允许你使用 `super` 。
- 构造器调用：产生一个 `TypeError` 。

在类定义中，名为 `constructor` 的方法很特别，在后面会有解释。

12.1.5 生成器方法定义

生成器方法在生成器章节介绍。它的语法和函数定义类似，但是带有一个额外的星号：

```
let obj = {
    * generatorMethod(···) {
        ···
    },
};
class MyClass {
    * generatorMethod(···) {
        ···
    }
}
```

规则：

- 调用一个生成器方法返回一个生成器对象。
- 你可以使用 `this` 和 `super` ，就像你在通常的方法定义中一样。

12.1.6 箭头函数

箭头函数在它自己的那一章讲解：

```
let squares = [1,2,3].map(x => x * x);
```

下面的变量词法上在箭头函数里（从父作用域中拉取）：

- `arguments`
- `super`
- `this`
- `new.target`

规则：

- 函数调用：词法的 `this` 等等。
- 方法调用：你可以使用箭头函数作为方法，但是 `this` 仍然是词法的，并不指向方法调用消息的接收者。
- 构造函数调用：产生一个 `TypeError` 。

12.1.7 类

类在它自己的那一章中讲解。

```
// Base class: no `extends`
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return `(${this.x}, ${this.y})`;
    }
}

// This class is derived from `Point`
class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    toString() {
        return super.toString() + ' in ' + this.color;
    }
}
```

`constructor` 方法很特殊，因为它“变成了”类。也就是说，类和构造器函数非常相似：

```
> Point.prototype.constructor === Point
true
```

规则：

- 函数/方法调用：类不能以函数或者方法的形式调用（在类的那一章中解释了为什么）。
- 构造器调用：遵循一个支持子类的协议。在基类中，创建一个实例，然后 `this` 指向它。一个继承类从它的父类中获取它的实例，这就是为什么要在访问 `this` 之前使用 `super` 。

12.2 风格思考

有些关于使用实体调用时的推荐。

12.2.1 推荐使用箭头函数作为回调

无论什么时候，只要可以，都应该使用箭头函数作为回调，而不是传统的方式。箭头函数更加方便，因为可以拿到词法的 `this` ，并且语法上更加紧凑。

12.2.1.1 问题：作为隐式参数的 `this`

不幸的是，一些 JavaScript API 使用 `this` 作为回调函数的隐式参数，此时就没法使用箭头函数了。例如：行 B 中的 `this` 是行 A 函数的隐式参数。

```
beforeEach(function () { // (A)
    this.addMatchers({ // (B)
        toBeInRange: function (start, end) {
            ...
        }
    });
});
```

12.2.1.2 解决方案1：修改 API

修复很简单，但是需要修改 API ：

```
beforeEach(api => {
    api.addMatchers({
        toBeInRange(start, end) {
            ...
        }
    });
});
```

我们将 API 从一个隐式的变量 `this` 转换成了显示的参数 `api` 。我喜欢这种明确性。

12.2.1.3 解决方案2：用另外的方式获取 `this` 的值

在一些 API 中，有一些可选的方式得到 `this` 值。例如，下面代码使用 `this` 。

```
var $button = $('#myButton');
$button.on('click', function () {
    this.classList.toggle('clicked');
});
```

但是这个事件的目标也可以通过 `event.target` 拿到：

```
var $button = $('#myButton');
$button.on('click', event => {
    event.target.classList.toggle('clicked');
});
```

12.2.2 当心函数定义

只要不访问 `this` ，函数定义和非方法函数一样安全。

```
function foo(arg1, arg2) {
    ...
}
```

也可以可选地使用 `const` 和箭头函数，这会得到词法的 `this` ，但是 - 可能 - 看起来不漂亮：

```
const foo = (arg1, arg2) => {
    ...
};
```

12.2.3 对于方法，推荐使用方法定义

方法定义是唯一的一种创建方法（使用 `super` ）的方式。这是对象字面量和类（定义方法的唯一方式）明显的选择，但是要给已经存在的对象添加一个方法会怎么样呢？例如：

```
MyClass.prototype.foo = function (arg1, arg2) {
    ...
};
```

下面是一个在 ES6 中完成同样事情的快速（某种程度上会造成污染）方式。

```
Object.assign(MyClass.prototype, {
    foo(arg1, arg2) {
        ...
    }
});
```

更多的信息和说明，参看关于 `Object.assign()` 的那一节。

12.2.4 方法和回调函数属性之间的区别

在对象方法和作为回调集合的对象之间有一些细微的差别。

例如，你可以像下面一样使用 WHATWG 流 API：

```
let stream = new ReadableStream({
    start(controller) {
        ...
    },

    pull() {
        ...
    },

    cancel() {
        ...
    }
});
```

也就是说， `ReadableStream()` 的参数是一个带有方法的对象。这允许 `start` ， `pull` 和 `cancel` 使用 `this` 来彼此调用，访问本地对象（ object-local ）状态。

另一方面，你可能轻易地看到这三个指向回调函数的属性，这些回调函数与父作用域（例如一个方法）交互。此时 `this` 应该就是词法范围的，代码看起来像下面这样。

```
let stream = new ReadableStream({
    start: controller => {
        ...
    },

    pull: () => {
        ...
    },

    cancel: () => {
        ...
    }
});
```

12.2.5 在 ES6 中避免 IIFE

本节给出在 ES6 中避免 IIFE 的小技巧。

12.2.5.1 用代码块替换 IIFE

在 ES5 中，如果想保持一个变量本地化，就必须要使用 IIFE ：

```
(function () {  // open IIFE
    var tmp = ···;
    ···
}());  // close IIFE
```

12.2.5.2 使用模块替换 IIFE

在 ECMAScript 5 代码中，并不会通过库（比如 RequireJS ， browserify 或者 webpack ）来使用模块，直接的模块形式很流行，并基于 IIFE 。它的好处就是清楚地区分了公开的和私有的：

```
var my_module = (function () {
    // Module-private variable:
    var countInvocations = 0;

    function myFunc(x) {
        countInvocations++;
        ...
    }

    // Exported by module:
    return {
        myFunc: myFunc
    };
}());
```

上例中的模块形式产生一个全局的变量，可以像下面这样使用：

```
my_module.myFunc(33);
```

在 ECMAScript 6 中，模块是内置的，这就是为什么采取这种方式的障碍很小：

```
// my_module.js

// Module-private variable:
let countInvocations = 0;

export function myFunc(x) {
    countInvocations++;
    ...
}
```

该模块不会产生一个全局的变量，使用方式如下：

```
import { myFunc } from 'my_module.js';

myFunc(33);
```

12.2.5.3 立即调用的箭头函数

在 ES6 中有一种场景仍然需要立即调用函数：有时候想通过一组语句产生一个结果，而不是通过一个单一的表达式。如果想内联这些语句，就必须要立即调用一个函数。在 ES6 中，如果想的话，可以使用立即调用的箭头函数：

```
const SENTENCE = 'How are you?';
const REVERSED_SENTENCE = (() => {
    // Iteration over the string gives us code points
    // (better for reversal than characters)
    let arr = [...SENTENCE];
    arr.reverse();
    return arr.join('');
})();
```

注意，必须像上面展示的一样加上小括号（小括号包住箭头函数，而不是整个函数调用）。在箭头函数那一章有详细的解释。

12.2.6 使用类

ES6 的类是不完美的，有一些批评者。但是我还是建议使用类，因为也有一些面向对象的论据对 ES6 的类有利，我在关于类的那一章中有介绍。

12.3 ECMAScript 5 和 6 中的传递方法调用消息和直接调用方法

在 JavaScript 中有两种方式调用方法：

- 通过发送调用消息，例如 `obj.someMethod(arg0, arg1)`
- 直接地，比如 `someFunc.call(thisValue, arg0, arg1)`

本节解释了这两种方式如何运作，和为什么在 ES6 中将会很少直接调用方法。在开始之前，先复习一下关于原型链的知识。

12.3.1 背景知识：原型链

记住 JavaScript 中每一个对象实际上都是在一条链上的，这条链上面有一个或多个对象。第一个对象从后一个对象上继承属性。例如，数组 `['a', 'b']` 的原型链看起来像这样：

- 1、存放元素 `'a'` 和 `'b'` 的实例
- 2、 `Array` 构造器提供的属性集 `Array.prototype`
- 3、 `Object` 构造器提供的属性集 `Object.prototype`
- 4、 `null` （链的末端，因此不是一个真正的成员）

可以通过 `Object.getPrototypeOf()` 查看原型链：

```
> var arr = ['a', 'b'];
> var p = Object.getPrototypeOf;

> p(arr) === Array.prototype
true
> p(p(arr)) === Object.prototype
true
> p(p(p(arr)))
null
```

“前面的”对象上的属性覆盖“后面”对象上的属性。例如， `Array.prototype` 提供了一个数组版本的 `toString()` 方法，覆盖了 `Object.prototype.toString()` 。

```
> var arr = ['a', 'b'];
> Object.getOwnPropertyNames(Array.prototype)
[ 'toString', 'join', 'pop', ··· ]
> arr.toString()
'a,b'
```

12.3.2 发送方法调用消息

方法调用 `arr.toString()` 实际上包含两个步骤：

- 1、消息发送：在 `arr` 的原型链上，找到第一个名字为 `toString` 的属性。
- 2、调用：调用找到的值，然后设置隐式的 `this` 参数为消息接收者 `arr` 。

你可以通过使用函数的 `call()` 方法显示地执行这两个步骤：

```
> var func = arr.toString; // dispatch
> func.call(arr) // direct call, providing a value for `this`
'a,b'
```

12.3.3 直接方法调用

在 JavaScript 中有两种方式直接调用方法：

- `Function.prototype.call(thisValue, arg0?, arg1?, ···)`
- `Function.prototype.apply(thisValue, argArray)`

`call` 和 `apply` 这两个方法都在函数上调用。它们是通常的函数调用不一样，因为可以指定 `this` 值。 `call` 在方法调用中一个个地提供参数， `apply` 是通过一个数组提供。

通过动态发送消息的方式调用方法有一个问题，方法需要在对象的原型链中。 `call()` 使得可以在调用一个方法的时候直接指定消息接收者。这意味着可以其它对象中借出一个不在当前原型链中的方法。例如，可以对 `arr` 调用原始的未被覆盖的 `Object.prototype.toString` 方法：

```
> Object.prototype.toString.call(arr)
'[object Array]'
```

对大量对象都有效的方法（不仅仅是“它们的”构造器生成的实例）称作通用方法（ generic ）。《 Speaking JavaScript 》中有一组方法全是通用方法，包含大多数数组方法和所有 `Object.prototype` （这上面的方法必须要能正确处理所有对象，因此隐式地就是通用方法了）上的方法。

12.3.4 直接方法调用的使用场景

本节涵盖了直接方法调用的使用场景。每一次，我会先描述 ES5 中的使用场景，然后是 ES6 中的变化（将会很少使用直接方法调用了）。

12.3.4.1 ES5 ：通过一个数组给一个方法提供参数

一些函数接收多个值，但是每个参数对应一个值。如果想通过一个数组传入这些值，怎么做？

例如， `push()` 给一个数组追加几个值：

```
> var arr = ['a', 'b'];
> arr.push('c', 'd')
4
> arr
[ 'a', 'b', 'c', 'd' ]
```

但是却不能追加整个数组。可以通过使用 `apply()` 来变通处理这个限制：

```
> var arr = ['a', 'b'];
> Array.prototype.push.apply(arr, ['c', 'd'])
4
> arr
[ 'a', 'b', 'c', 'd' ]
```

类似地， `Math.max()` 和 `Math.min()` 仅对单一值有效：

```
> Math.max(-1, 7, 2)
7
```

有了 `apply()` ，可以用数组的方式来使用这些方法：

```
> Math.max.apply(null, [-1, 7, 2])
7
```

12.3.4.2 ES6 ：扩展操作符（...）基本上取代了 `apply()`

通过 `apply()` 的直接的方法调用，仅仅是因为像把一个数组转换成一组参数显得很笨拙，这就是为什么 ECMAScript 6 有扩展操作符（...）。甚至在发送方法调用消息中也可以使用。

```
> Math.max(...[-1, 7, 2])
7
```

另一个例子：

```
> let arr = ['a', 'b'];
> arr.push(...['c', 'd'])
4
> arr
[ 'a', 'b', 'c', 'd' ]
```

扩展操作符对 `new` 操作符同样有效：

```
> new Date(...[2011, 11, 24])
Sat Dec 24 2011 00:00:00 GMT+0100 (CET)
```

注意 `apply()` 不能和 `new` 结合使用 - 上述功能在 ECMAScript 5 中只能通过复杂的迂回方式来实现。

12.3.4.3 ES5：把类数组对象转换成数组

JavaScript 中的一些对象是类数组的，它们很像数组了，但是没有任何数组方法。看两个例子。

第一个，函数中特殊的变量 `arguments` 就是类数组的。它有一个 `length` 属性和索引化的元素访问。

```
> var args = function () { return arguments }('a', 'b');
> args.length
2
> args[0]
'a'
```

但是 `arguments` 并不是 `Array` 的实例，没有 `forEach()` 方法。

```
> args instanceof Array
false
> args.forEach
undefined
```

第二个，DOM 方法 `document.querySelectorAll()` 返回一个 `NodeList` 的实例。

```
> document.querySelectorAll('a[href]') instanceof NodeList
true
> document.querySelectorAll('a[href]').forEach // no Array metho
ds!
undefined
```

对于很多复杂的操作，需要先将类数组对象转换成数组。这通过 `Array.prototype.slice()` 实现。该方法将方法调用接收者的元素拷贝到一个新的数组里面：

```
> var arr = ['a', 'b'];
> arr.slice()
[ 'a', 'b' ]
> arr.slice() === arr
false
```

如果直接调用 `slice()` ，可以将一个 `NodeList` 转换成数组：

```
var domLinks = document.querySelectorAll('a[href]');
var links = Array.prototype.slice.call(domLinks);
links.forEach(function (link) {
    console.log(link);
});
```

也可以将 `arguments` 转换成数组：

```
function format(pattern) {
    // params start at arguments[1], skipping `pattern`
    var params = Array.prototype.slice.call(arguments, 1);
    return params;
}
console.log(format('a', 'b', 'c')); // ['b', 'c']
```

12.3.4.4 ES6：类数组对象不再恼人

一方面，ECMAScript 6 有 `Array.from()` ，一个简单地将类数组对象转换成数组的方法：

```
let domLinks = document.querySelectorAll('a[href]');
let links = Array.from(domLinks);
links.forEach(function (link) {
    console.log(link);
});
```

另一方面，不会再需要类数组的 `arguments` ，因为 ECMAScript 6 有剩余参数（通过三个点声明）：

```
function format(pattern, ...params) {
    return params;
}
console.log(format('a', 'b', 'c')); // ['b', 'c']
```

12.3.4.5 ES5 ：安全地使用 `hasOwnProperty()`

`obj.hasOwnProperty('prop')` 可以辨别 `obj` 是否有自有（不是继承来的）属性 `prop` 。

```
> var obj = { prop: 123 };

> obj.hasOwnProperty('prop')
true

> 'toString' in obj // inherited
true
> obj.hasOwnProperty('toString') // own
false
```

然而，如果覆盖了 `Object.prototype.hasOwnProperty` ，那么通过发送方法调用消息的方式调用 `hasOwnProperty` 将会得到错误的结果。

```
> var obj1 = { hasOwnProperty: 123 };
> obj1.hasOwnProperty('toString')
TypeError: Property 'hasOwnProperty' is not a function
```

如果 `Object.prototype` 没有在对象的原型链上， `hasOwnProperty` 就不能通过发送消息的方式调用了。

```
> var obj2 = Object.create(null);
> obj2.hasOwnProperty('toString')
TypeError: Object has no method 'hasOwnProperty'
```

在这两种情形下，解决的方法是使用直接调用的方式：

```
> var obj1 = { hasOwnProperty: 123 };
> Object.prototype.hasOwnProperty.call(obj1, 'hasOwnProperty')
true

> var obj2 = Object.create(null);
> Object.prototype.hasOwnProperty.call(obj2, 'toString')
false
```

12.3.4.6 ES6 ：很少使用 `hasOwnProperty()`

`hasOwnProperty()` 大多数时候用于通过对象实现 Map 。所幸的是，ECMAScript 6 有内置的 `Map` 数据结构，这意味着将会很少使用 `hasOwnProperty()` 。

12.3.4.7 ES5 ：避免中间对象

在字符串上调用数组特有的方法（比如 `join()` ）一般都包含两个步骤：

```
var str = 'abc';
var arr = str.split(''); // step 1
var joined = arr.join('-'); // step 2
console.log(joined); // a-b-c
```

字符串是类数组的，可以成为通用数组方法的 `this` 值。因此，直接调用的话就可以少掉第一步：

```
var str = 'abc';
var joined = Array.prototype.join.call(str, '-');
```

类似地，在拆分（ split ）了字符串之后或者通过直接调用，都可以在字符串上调用 `map()` ：

```
> function toUpper(x) { return x.toUpperCase() }
> 'abc'.split('').map(toUpper)
[ 'A', 'B', 'C' ]

> Array.prototype.map.call('abc', toUpper)
[ 'A', 'B', 'C' ]
```

注意直接的调用可能更加高效，也更优雅，这种方式是很值得的！

12.3.4.8 ES6 ：避免中间对象

如果第二个参数传入回调函数， `Array.from()` 可以在一步之内执行转换和 map 操作。

```
> Array.from('abc', ch => ch.toUpperCase())
[ 'A', 'B', 'C' ]
```

提示一下，两步完成的方法是：

```
> 'abc'.split('').map(function (x) { return x.toUpperCase() })
[ 'A', 'B', 'C' ]
```

12.3.5 `Object.prototype` 和 `Array.prototype` 的缩写

你可以通过一个空的对象字面量（它的原型是 `Object.prototype` ）来访问 `Object.prototype` 上面的方法。例如，下面两个直接方法调用是等价的：

```
Object.prototype.hasOwnProperty.call(obj, 'propKey')
{}.hasOwnProperty.call(obj, 'propKey')
```

对于 `Array.prototype` 也是同样的：

```
Array.prototype.slice.call(arguments)
[].slice.call(arguments)
```

这种方式很流行。相对于更长那种方式，短的那种方式并没有很清楚地反映代码书写者的意图，但是没那么啰嗦。执行速度上面，两种版本都没有太大区别。

13 箭头函数

13.1 概览

箭头函数有两个好处。

第一个，没有传统函数那么啰嗦：

```
let arr = [1, 2, 3];
let squares = arr.map(x => x * x);

// Traditional function expression:
let squares = arr.map(function (x) { return x * x });
```

第二个，`this` 从父作用域（词法的）中拉取。因此，不再需要使用 `bind()` 或者 `that = this` 。

```
function UiComponent {
    let button = document.getElementById('myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick(); // lexical `this`
    });
}
```

下面的变量都是词法范围的：

- `arguments`
- `super`
- `this`
- `new.target`

13.2 因为 `this` ，传统的函数都是糟糕的非方法函数

在 JavaScript 中，传统的函数可以被用作：

- 1、非方法函数
- 2、方法
- 3、构造器

它们之间是冲突的：因为在第二种和第三种中，函数总是有自己的 `this` 变量。但是，有时会拿不到想要的那个 `this` ，比如从一个回调函数（第一种）中获取外面方法中的 `this` 。

可以在下面的 ES5 代码中看到这个现象：

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) { // (A)
    'use strict';
    return arr.map(function (x) { // (B)
        // Doesn’t work:
        return this.prefix + x; // (C)
    });
};
```

在 C 行，我们想访问 `this.name` ，但是由于行 B 的函数在作用域链上覆盖了来自于行 A 方法的 `this` ，因此是无法访问的。在严格模式下， 非方法函数中的 `this` 是 `undefined` ，这就是为什么在使用 `Prefixer` 的时候抛出错误：

```
> var pre = new Prefixer('Hi ');
> pre.prefixArray(['Joe', 'Alex'])
TypeError: Cannot read property 'prefix' of undefined
```

在 ECMAScript 5 中有三种方法解决这个问题。

13.2.1 方法1： `that = this`

你可以将 `this` 赋值给一个不会被隐藏的变量。这就是下面行 A 所做的：

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    var that = this; // (A)
    return arr.map(function (x) {
        return that.prefix + x;
    });
};
```

现在 `Prefixer` 像预期一样运作了：

```
> var pre = new Prefixer('Hi ');
> pre.prefixArray(['Joe', 'Alex'])
[ 'Hi Joe', 'Hi Alex' ]
```

13.2.2 方法2：为 `this` 指定一个值

有几个数组方法有一个额外的参数用于指定在调用回调函数的时候的 `this` 值。这就是下面行 A 最后一个参数：

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    return arr.map(function (x) {
        return this.prefix + x;
    }, this); // (A)
};
```

13.2.3 方法3： `bind(this)`

可以使用方法 `bind()` 转换一下在调用的时候才决定的 `this` 值（通过 `call()` ，函数调用，方法调用，等等），将其变为固定的值。这就是下面行 A 所做的。

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    return arr.map(function (x) {
        return this.prefix + x;
    }.bind(this)); // (A)
};
```

13.2.4 ECMAScript 6 的解决方案：箭头函数

箭头函数基于方法3，用一种更加方便的语法。使用箭头函数，代码看起来像下面这样。

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    return arr.map((x) => {
        return this.prefix + x;
    });
};
```

如果要把上述代码完全 ES6 化，就要使用类和更加紧凑多样的箭头函数：

```
class Prefixer {
    constructor(prefix) {
        this.prefix = prefix;
    }
    prefixArray(arr) {
        return arr.map(x => this.prefix + x); // (A)
    }
}
```

在行 A ，调整了一下箭头函数两边的部分，节省了几个字符：

- 如果只有一个参数，并且这个参数是一个标识符，那么可以省略小括号。
- 在箭头后面跟一个表达式会使表达式的值被直接返回。

在上面的代码中，也可以看到方法 `constructor` 和 `prefixArray` 使用新的，更加紧凑的 ES6 语法定义，该种写法对对象字面量也同样有效。

13.3 箭头函数语法

选用“胖”箭头 `=>` （相对于瘦箭头 `->` ）的原因是与 CoffeeScript 兼容，两者非常类似。

指定参数：

```
    () => { ... } // no parameter
     x => { ... } // one parameter, an identifier
(x, y) => { ... } // several parameters
```

指定函数体：

```
x => { return x * x }  // block
x => x * x  // expression, equivalent to previous line
```

这些语句块表现得像普通的函数体。例如，需要用 `return` 返回一个值。如果只有一个表达式，那么这个表达式的值会被隐式返回。

注意，只有一个表达式的箭头函数省略了多少啰嗦的东西。比较：

```
let squares = [1, 2, 3].map(function (x) { return x * x });
let squares = [1, 2, 3].map(x => x * x);
```

13.4 词法范围的变量

13.4.1 变量值的来源：静态的和动态的

下面是一个变量能够接收到值的两种方式。

第一种，静态地（词法地）：变量的值由程序的书写解构决定，从父作用域中获取值。例如：

```
let x = 123;

function foo(y) {
    return x; // value received statically
}
```

第二种，动态地：通过函数调用获得值。例如：

```
function bar(arg) {
    return arg; // value received dynamically
}
```

13.4.2 箭头函数中的词法变量

`this` 的来源是一个区分箭头函数的重要方面：

- 传统函数有一个动态的 `this` ，它的值取决于函数如何调用。
- 箭头函数有一个词法的 `this` ，它的值取决于父作用域。

完整的从词法范围获取值的一组变量是：

- `arguments`
- `super`
- `this`
- `new.target`

13.5 语法陷阱

有几个语法相关的细节问题可能会给你带来麻烦。

13.5.1 箭头函数绑得非常松散

箭头函数绑得非常松散。原因是希望每一个能在表达式体中出现的表达式都“紧紧结合在一起”，但是实际上另一个表达式比箭头函数结合性更高。

结果，在其它某些地方，经常不得不将箭头函数包裹在小括号中。例如：

```
console.log(typeof () => {}); // SyntaxError
console.log(typeof (() => {})); // OK
```

另一方面，可以使用 `typeof` 作为一个表达式体，不用放在大括号中：

```
const f = x => typeof x;
```

13.5.2 立即调用的箭头函数

还记得立即调用函数表达式（ IIFEs ）吗？在 ECMAScript 5 中用于模拟块级作用域和值返回块，看起来像下面这样：

```
(function () { // open IIFE
    // inside IIFE
}()); // close IIFE
```

如果使用立即调用的箭头函数（ IIAF ），可以省掉几个字符：

```
(() => {
    return 123
})();
```

和 `IIFEs` 类似，应该在 IIAFs 结尾加上分好（或者使用一个等价的措施），以避免两个连续的 IIAFs 被解释成一个函数调用（第一个是函数，第二个是参数）。

即便 IIAF 有一个块体，也必须使用小括号包裹起来，因为它结合松散，不能（直接地）做函数调用。注意小括号一定是包住了箭头函数。使用 IIFEs 的话，你有一个选择：用小括号要么包住整个语句，要么包住函数表达式。

就像上节提到的，箭头函数结合松散，这对于表达式体很有用，比如你想让这种表达式：

```
const value = () => foo()
```

解释成这样：

```
const value = () => (foo())
```

而不是这样：

```
const value = (() => foo)()
```

本章关于可调用实体的那一节有更多关于在 ES6 中使用 IIFEs 和 IIFAs 的信息。

13.5.4 不能用语句作为表达式体

13.5.4.1 表达式与语句

快速复习（参阅《Speaking JavaScript》获取更多相关信息）：

表达式产生（执行得到）值。例子：

```
3 + 4
foo(7)
'abc'.length
```

语句完成功能。例子：

```
while (true) { ··· }
return 123;
```

大多数表达式可以被用作语句，简单地将它们放在语句的位置上：

```
function bar() {
    3 + 4;
    foo(7);
    'abc'.length;
}
```

13.5.4.2 箭头函数体

如果箭头函数体是一个表达式，那么就可以不需要括号了：

```
asyncFunc.then(x => console.log(x));
```

但是，语句一定要放在括号里面：

```
asyncFunc.catch(x => { throw x });
```

13.5.5 返回一个对象字面量

有一个块体而不是一个表达式体意味着如果你想让表达式体是一个对象字面量，就必须将其放在括号中。

箭头函数体是一个带有 `bar` 的标签和 `123` 的表达式语句。

```
let f = x => { bar: 123 }
```

箭头函数体是一个表达式，一个对象字面量：

```
let f = x => ({ bar: 123 })
```

13.6 箭头函数与常规函数对比

一个箭头函数与一个普通的函数在两个方面不一样：

- 下列变量的构造是词法的： `arguments` ， `super` ， `this` ， `new.target`
- 不能被用作构造函数：没有内部方法 `[[Construct]]` （该方法允许普通的函数通过 `new` 调用），也没有 `prototype` 属性。因此， `new (() => {})` 会抛出错误。

除了那些意外，箭头函数和普通的函数没有明显的区别。例如， `typeof` 和 `instanceof` 产生同样的结果：

```
> typeof () => {}
'function'
> () => {} instanceof Function
true

> typeof function () {}
'function'
> function () {} instanceof Function
true
```

参阅可调用实体的那一章，获取更多关于什么时候使用箭头函数和什么时候使用传统函数的信息。

函数表达式和对象字面量是例外，这种情形下必须放在括号里面，因为它们看起来像是函数声明和代码块。

14 除类之外的新的面向对象特性

类（在下一章讲解）是 ECMAScript 6 中主要的新的 OOP 特性。但是，也有一些对象字面量的新特性，和 `Object` 上的新的实用方法，本章将会讲解这些内容。

14.1 概览

14.1.1 新的对象字面量特性

方法定义：

```
let obj = {
    myMethod(x, y) {
        ...
    }
};
```

属性值缩写：

```
let first = 'Jane';
let last = 'Doe';

let obj = { first, last };
// Same as:
let obj = { first: first, last: last };
```

计算属性键：

```
let propKey = 'foo';
let obj = {
    [propKey]: true,
    ['b'+'ar']: 123
};
```

这种新语法也可用于方法定义：

```
let obj = {
    ['h'+'ello']() {
        return 'hi';
    }
};
console.log(obj.hello()); // hi
```

计算属性键主要的应用场景就是使 symbol 成为属性键变得更加方便。

14.1.2 `Object` 中的新方法

`Object` 中最重要的新方法是 `assign()` 。习惯上，在 JavaScript 的世界中，这个函数叫做 `extend()` 。 `Object.assign()` 仅考虑自有（非继承）属性。

```
let obj = { foo: 123 };
Object.assign(obj, { bar: true });
console.log(JSON.stringify(obj));
    // {"foo":123,"bar":true}
```

14.2 对象字面量的新特性

14.2.1 方法定义

在 ECMAScript 5 中，方法是值为函数的属性：

```
var obj = {
    myMethod: function (x, y) {
        ...
    }
};
```

在 ECMAScript 6 中，方法仍然是函数值属性，但是现在多了一种定义方法的方式：

```
let obj = {
    myMethod(x, y) {
        ...
    }
};
```

getters 和 setters 仍然和 ECMAScript 5 中一样有效（注意在语法上和方法定义是如何相似的）：

```
let obj = {
    get foo() {
        console.log('GET foo');
        return 123;
    },
    set bar(value) {
        console.log('SET bar to '+value);
        // return value is ignored
    }
};
```

使用 `obj` ：

```
> obj.foo
GET foo
123
> obj.bar = true
SET bar to true
true
```

也有一种简明的方式定义值为生成器函数的属性：

```
let obj = {
    * myGeneratorMethod() {
        ...
    }
};
```

上述代码等价于：

```
let obj = {
    myGeneratorMethod: function* () {
        ...
    }
};
```

14.2.2 属性值缩写

属性值简称使你在对象字面量中简写属性的定义：如果用于指定属性值的变量名字也是属性键，则可以省略键。看起来像下面这样：

```
let x = 4;
let y = 1;
let obj = { x, y };
```

最后一行等价于：

```
let obj = { x: x, y: y };
```

属性值简称也可以用于解构：

```
let obj = { x: 4, y: 1 };
let {x,y} = obj;
console.log(x); // 4
console.log(y); // 1
```

属性值简称的使用场景之一就是多值返回（在解构那一章讲解了）。

14.2.3 计算的属性键

记住在设置属性的时候，有两种方法指定一个键。

- 1、通过一个固定的名字： `obj.foo = true` ；

- 2、通过表达式： `obj['b' + 'ar']` = 123；

在对象字面量中， ECMAScript 5 只能使用第一种方式。 ECMAScript 6 提供了可选的第二种方式：

```
let propKey = 'foo';
let obj = {
    [propKey]: true,
    ['b'+'ar']: 123
};
```

新的语法也可以用于方法定义：

```
let obj = {
    ['h'+'ello']() {
        return 'hi';
    }
};
console.log(obj.hello()); // hi
```

计算属性键的主要使用场景是 Symbol ：可以定义一个公开的 symbol ，然后使用它作为特殊的属性键，该键总是唯一的。一个出名的例子就是存储在 `Symbol.iterator` 中的 Symbol 。如果某个对象有一个方法的键是 `Symbol.iterator` ，那么这个对象就变成了可迭代的对象：这个方法必须返回一个迭代器，该迭代器用于在 `for-of` 循环等地方迭代这个对象。下面的代码展示了这是怎么运作的。

```
let obj = {
    * [Symbol.iterator]() { // (A)
        yield 'hello';
        yield 'world';
    }
};
for (let x of obj) {
    console.log(x);
}
// Output:
// hello
// world
```

行 A 以一个计算键（存储在 `Symbol.iterator` 中的 Symbol ）开始生成器函数的定义。

14.3 新的 Object 方法

14.3.1 `Object.assign(target, source_1, source_2, ···)`

该方法将源数据（source）合并到目标数据（target）中去：会修改 `target`，第一步拷贝 `source_1` 上所有的可枚举自有属性到 `target`，然后是 `source_2`，以此类推。最后，返回目标数据。

```
let obj = { foo: 123 };
Object.assign(obj, { bar: true });
console.log(JSON.stringify(obj));
    // {"foo":123,"bar":true}
```

让我们更近距离地看下 `Object.assign()` 是如何运作的：

- 两种属性键：`Object.assign()` 支持字符串和 Symbol 作为属性键。
- 仅操作可枚举的自有属性：`Object.assign()` 忽略继承的属性和不可枚举的属性。
- 通过赋值来拷贝：在目标对象中的属性通过赋值（内部操作 [[Put]]）来创建。这意味着如果 `target` 上有（自有的或继承的）setters，在拷贝的时候会被调用。另一种可选的定义新属性的方式就是始终创建新的属性，而不调用 setters。本来有人提议让 `Object.assign()` 也支持使用定义而不是赋值的方式，这个建议最终在 ECMAScript 6 中被拒绝了，但是可能在后面的版本中会重新考虑。
- 不能移动使用 `super` 的方法：这样的方法有一个内部属性 `[[HomeObject]]`，该属性将该方法与创建该方法的时候所在的对象绑定起来。如果通过 `Object.assign()` 移动这个方法，该方法将会保持和原对象的这种绑定。详细内容在本章关于类的那一节讲解。

14.3.1.1 `Object.assign()` 的使用场景

让我们看几个使用场景。

14.3.1.1.1 给 `this` 添加属性

在构造器中可以使用 `Object.assign()` 给 `this` 添加属性：

```
class Point {
    constructor(x, y) {
        Object.assign(this, {x, y});
    }
}
```

14.3.1.1.2 给对象属性提供默认值

`Object.assign()` 在给缺失的属性填充默认值的时候也很有用。在下面的例子中，有一个带有默认值的对象 `DEFAULTS` 和一个带有数据的对象 `options` 。

```
const DEFAULTS = {
    logLevel: 0,
    outputFormat: 'html'
};
function processContent(options) {
    options = Object.assign({}, DEFAULTS, options); // (A)
    ...
}
```

在行 A ，创建了一个新的对象，把默认属性拷贝过来，然后把 `options` 上面的属性拷贝过来，覆盖默认值。 `Object.assign()` 返回这些操作的结果，然后赋值给 `options` 。

14.3.1.1.3 给对象添加方法

另一个使用场景就是给对象添加方法：

```
Object.assign(SomeClass.prototype, {
    someMethod(arg1, arg2) {
        ...
    },
    anotherMethod() {
        ...
    }
});
```

也可以手动地将函数赋值上去，但是这样写出来的代码就没那么漂亮了，并且每次都需要写一遍 `SomeClass.prototype` ：

```
SomeClass.prototype.someMethod = function (arg1, arg2) {
    ...
};
SomeClass.prototype.anotherMethod = function () {
    ...
};
```

14.3.1.1.4 克隆对象

最后一个 `Object.assign()` 的使用场景就是快速克隆对象：

```
function clone(orig) {
    return Object.assign({}, orig);
}
```

这种克隆方式某种程度上来讲是不完美的，因为并不会保持 `orig` 的属性特性（可写、可枚举等）。如果这是你想要的，就必须使用属性描述器（ property descriptors ）了。

如果想让克隆出来的对象和原对象有相同的原型，可以使用 `Object.getPrototypeOf()` 和 `Object.create()` ：

```
function clone(orig) {
    let origProto = Object.getPrototypeOf(orig);
    return Object.assign(Object.create(origProto), orig);
}
```

14.3.2 `Object.getOwnPropertySymbols(obj)`

`Object.getOwnPropertySymbols(obj)` 获取 `obj` 上所有自有的为 Symbol 的键。它补充了 `Object.getOwnPropertyNames()` ，这个方法获取自有的字符串形式的键。后面的一节中详细讲解了迭代属性键的知识。

14.3.3 Object.is(value1, value2)

严格相等操作符（===）处理两个值的结果可能不是想象中的那样。

首先， `NaN` 和自身不相等。

```
> NaN === NaN
false
```

这很不幸，因为这经常阻止我们检测 `NaN` ：

```
> [0,NaN,2].indexOf(NaN)
-1
```

其次， JavaScript 有两种零值，但是在严格模式下处理结果看起来似乎是同一个值：

```
> -0 === +0
true
```

这样做通常情况下是好的。

`Object.is()` 提供了一种相对于 `===` 明确一点的比较值的方式。如下所示：

```
> Object.is(NaN, NaN)
true
> Object.is(-0, +0)
false
```

其它所有的比较结果都和 `===` 一样。

14.3.3.1 使用 `Object.is()` 查找数组元素

如果将 `Object.is()` 和新的 ES6 数组方法 `findIndex()` 结合起来，可以找到数组中的 `NaN` ：

```
function myIndexOf(arr, elem) {
    return arr.findIndex(x => Object.is(x, elem));
}

myIndexOf([0,NaN,2], NaN); // 1
```

相比之下， `indexOf()` 并不能很好地处理 `NaN` ：

```
> [0,NaN,2].indexOf(NaN)
-1
```

14.3.4 `Object.setPrototypeOf(obj, proto)`

该方法将 `obj` 的原型设置为 `proto` 。在 ECMAScript 5 中有一种非标准的方式，这种方式被很多引擎支持，就是通过给特殊的属性 `__proto__` 赋值。推荐的设置原型的方式还是与在 ECMAScript 5 中使用的一样：通过 `Object.create()` 创建对象。这总是比先创建对象再设置原型的方式快。很明显，这种方式在想要改变已有对象的原型的时候是不行的。

14.4 ES6 中遍历属性键

在 ECMAScript 5 中，一个属性的键可以是字符串或者 Symbol 。现在有五种工具方法从 `obj` 对象中获取属性键：

- `Object.keys(obj) : Array<string>`

  获取所有可枚举的自有（非继承）属性的字符串键。

- `Object.getOwnPropertyNames(obj) : Array<string>`

  获取所有自有属性的字符串键。

- `Object.getOwnPropertySymbols(obj) : Array<symbol>`

  获取所有自有属性的 Symbol 键。

- `Reflect.ownKeys(obj) : Array<string|symbol>`

  获取所有自有属性的键。

- `Reflect.enumerate(obj) : Iterator`

  获取所有可枚举属性的字符串键。

14.4.1 属性键的迭代顺序

所有迭代属性键的方法都是一样的顺序：

- 首先是所有的数组索引，按照数字排序。
- 然后是所有字符串键（非索引），按照创建的顺序。
- 然后是所有 Symbol ，按照创建的顺序。

注意，语言规范把索引定义为字符串键，当转换成无符号整数然后再转换回来，仍然是同样的字符串（同样也必须比2^{32}-1小，因此数组是有长度限制的）。这意味着规范把数组索引看作字符串键，甚至普通的对象都可以有数组索引：

```
> Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })
['2', '10', 'b', 'a', Symbol()]
```

> 为什么规范要标准化属性键返回的顺序？
>
> Tab Atkins Jr. 的回答：
>
> 因为，至少对于对象，所有在使用中的实现返回的顺序大致和当前的规范一样，并且大量的代码无形中就写成了依赖这种顺序的形式，如果用不同的顺序来枚举的话，将会造成破坏。既然浏览器必须要实现这种特定的顺序来变得 web 兼容，那么规范就要做这种顺序要求。
>
> 有一些关于从 Map/Set 中打破这种顺序的讨论，但是这样做的话就需要我们指定一种代码无法依赖的顺序；换句话说，我们必须让顺序是随机的，而不是非指定的。这要做更多的努力，并且创建顺序是很合理的（例如，参考 Python 中的 OrderedDict ），因此决定让 Map 和 Set 和 Object 一样。
>
> 规范中有两部分和本节有关：
>
> - 关于带有数组感觉的对象那一节提到了什么是数组索引。
> - 关于内部方法 [[OwnPropertyKeys]] 的那一节定义了属性返回的顺序。

14.5 常见问题的解答：对象字面量

14.5.1 可以再对象字面量中使用 `super` 吗？

是的，可以！详细内容在关于类的那一章讲解。

对象的自有属性就是那些不是从原型链中继承的属性。

15 类

本章讲解了 ES6 的类是如何运作的。

15.1 概览

类和子类：

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return `(${this.x}, ${this.y})`;
    }
}

class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    toString() {
        return super.toString() + ' in ' + this.color;
    }
}
```

在背后， ES6 类不是一个全新的东西：主要提供了更多方便的语法去创建老式的构造器函数。如果使用 `typeof` 可以看到：

```
> typeof Point
'function'
```

使用类：

```
> let cp = new ColorPoint(25, 8, 'green');

> cp.toString();
'(25, 8) in green'

> cp instanceof ColorPoint
true
> cp instanceof Point
true
```

15.2 要点

15.2.1 基类

在 ECMAScript 6 （ ES6 ）中，类像这样定义：

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return `(${this.x}, ${this.y})`;
    }
}
```

这个类使用起来就像一个 ES5 的构造器函数一样：

```
> var p = new Point(25, 8);
> p.toString()
'(25, 8)'
```

实际上，类定义的结果就是一个函数：

```
> typeof Point
'function'
```

然而，只能通过 `new` 调用类，不能通过函数调用（原理在后面解释）：

```
> Point()
TypeError: Classes can’t be function-called
```

> 在规范中，类以函数调用方式使用的时候，会被函数对象内部的方法 [[Call]] 阻止。

15.2.1.1 类声明不会被提升

函数声明会被提升：当进入一个作用域的时候，声明在里面的函数马上就可用了 - 不管函数声明在哪个位置。这意味着可以在函数声明之前调用：

```
foo(); // works, because `foo` is hoisted

function foo() {}
```

相对地，类声明不会提升。因此，仅在执行到类定义的地方，并且执行完类定义代码，类才会存在。在类声明之前访问类会抛出 `ReferenceError` 错误：

```
new Foo(); // ReferenceError

class Foo {}
```

这种限制的原因是类可以有一个 `继承` 子句，子句的值是任意表达式。这个表达式必须在正确的“地方”被执行，它的执行不能被提升。

没有提升功能导致的限制可能比你想象要少。例如，在类声明前面的函数依然能够访问那个类，但是必须要等到类声明已经被执行掉之后才能调用这个函数。

```
function functionThatUsesBar() {
    new Bar();
}

functionThatUsesBar(); // ReferenceError
class Bar {}
functionThatUsesBar(); // OK
```

15.2.1.2 类表达式

与函数类似，有两种类定义，两种方式定义一个类：类声明和类表达式。

同样地类似于函数，类表达式的标识符仅在当前表达式中可见：

```
const MyClass = class Me {
    getClassName() {
        return Me.name;
    }
};
let inst = new MyClass();
console.log(inst.getClassName()); // Me
console.log(Me.name); // ReferenceError: Me is not defined
```

15.2.2 在类定义的主体内部

类的主体只能包含方法，不能有数据属性。原型上面的数据属性一般会被认为是反模式，因此这种做法仅强制执行了一种最佳实践。

15.2.2.1 `构造器` ，静态方法，原型方法

让我们检测一下类定义中常见的三种方法：

```
class Foo {
    constructor(prop) {
        this.prop = prop;
    }
    static staticMethod() {
        return 'classy';
    }
    prototypeMethod() {
        return 'prototypical';
    }
}
let foo = new Foo(123);
```

该类声明的对象图看起来像下面这样。理解此图的小提示： `[[Prototype]]` 在对象之间是继承关系， `prototype` 是一个普通的值为对象的属性。属性 `prototype` 仅仅是比较特殊，因为 `new` 操作符使用它的值作为新创建实例的原型。



首先，伪方法 **`constructor`** 。这个方法很特殊，它定义了代表这个类的函数：

```
> Foo === Foo.prototype.constructor
true
> typeof Foo
'function'
```

有时称它为 `类构造器` 。它有一些普通构造器函数所不具备的特性（主要是能通过 `super()` 调用父构造器，这在后面讲解）。

其次，静态方法。静态属性（或者说是类属性）就是 `Foo` 自身上面的属性。如果定义方法的时候在前面加上 `static` ，就创建了一个类方法：

```
> typeof Foo.staticMethod
'function'
> Foo.staticMethod()
'classy'
```

第三点，原型方法。 `Foo` 的原型属性就是 `Foo.prototype` 的属性。它们是常用的方法，并被 `Foo` 的实例继承。

```
> typeof Foo.prototype.prototypeMethod
'function'
> foo.prototypeMethod()
'prototypical'
```

15.2.2.2 静态数据属性

到目前为止，类只允许创建静态方法，不能创建静态数据属性。有两种方法可以解决这个问题。

第一个，可以手动地添加一个静态属性：

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
Point.ZERO = new Point(0, 0);
```

第二个，创建一个静态的 getter ：

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    static get ZERO() {
        return new Point(0, 0);
    }
}
```

在两种场景中，都得到一个能读取的属性 `Point.ZERO` 。在第一种方法中，可以使用 `Object.defineProperty()` 创建一个只读的属性，但是我喜欢赋值的简洁性。

15.2.2.3 getters 和 setters

getters 和 setters 的语法和 ECMAScritp 5 中的对象字面量语法是类似的：

```
class MyClass {
    get prop() {
        return 'getter';
    }
    set prop(value) {
        console.log('setter: '+value);
    }
}
```

你可以像下面这样使用 `MyClass` 。

```
> let inst = new MyClass();
> inst.prop = 123;
setter: 123
> inst.prop
'getter'
```

15.2.2.4 计算的方法名字

可以通过表达式定义方法名字，需要把表达式放在一对中括号中。例如，下面定义 `Foo` 的所有方式都是等价的。

```
class Foo() {
    myMethod() {}
}

class Foo() {
    ['my'+'Method']() {}
}

const m = 'myMethod';
class Foo() {
    [m]() {}
}
```

ECMAScript 6 中有几个特殊的方法，这些方法的键是 Symbol 。计算方法名字让你能够定义这种方法。例如，如果一个对象有一个方法，这个方法的键是 `Symbol.iterator` ，那么这个对象就是可迭代的。这意味着它的内容可以通过 `for-of` 循环和其它语言机制来迭代。

```
class IterableClass {
    [Symbol.iterator]() {
        ...
    }
}
```

15.2.2.5 生成器函数

如果在方法定义的时候加上一个星号前缀，那么这个方法就会变成生成器方法。在其它方面，生成器对于定义键是 `Symbol.iterator` 的方法是很有用的。下面的代码展示了如何运作。

```
class IterableArguments {
    constructor(...args) {
        this.args = args;
    }
    * [Symbol.iterator]() {
        for (let arg of this.args) {
            yield arg;
        }
    }
}

for (let x of new IterableArguments('hello', 'world')) {
    console.log(x);
}

// Output:
// hello
// world
```

15.2.3 子类

继承子句让你能够创建已有构造器（这个构造器可能是通过类的形式创建的，也可能不是）的子类：

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return `(${this.x}, ${this.y})`;
    }
}

class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y); // (A)
        this.color = color;
    }
    toString() {
        return super.toString() + ' in ' + this.color; // (B)
    }
}
```

该类的使用和想象的一样：

```
> let cp = new ColorPoint(25, 8, 'green');
> cp.toString()
'(25, 8) in green'

> cp instanceof ColorPoint
true
> cp instanceof Point
true
```

有两种类：

- `Point` 是一个基类，因为它没有继承子句。
- `ColorPoint` 是一个继承类。

有两种使用 `super` 的方式：

- 在类构造器（类定义中的伪方法 `constructor` ）中使用它就像方法调用一样（ `super(...)` ），用于调用父构造器（行 A ）。
- 在方法定义（在对象字面量或者类中，带有或者不带 `static` ）中使用它就像属性引用（ `super.prop` 或者方法调用（ `super.method(...)` ）），用于访问父属性（行 B ）。

15.2.3.1 子类的原型是父类

在 ECMAScript 6 中，子类的原型是父类：

```
> Object.getPrototypeOf(ColorPoint) === Point
true
```

这意味着静态属性是会被继承的：

```
class Foo {
    static classMethod() {
        return 'hello';
    }
}

class Bar extends Foo {
}
Bar.classMethod(); // 'hello'
```

甚至可以通过 `super` 调用父类中的静态方法：

```
class Foo {
    static classMethod() {
        return 'hello';
    }
}

class Bar extends Foo {
    static classMethod() {
        return super.classMethod() + ', too';
    }
}
Bar.classMethod(); // 'hello, too'
```

15.2.3.2 父构造器调用

在一个继承类里面， `super()` 调用必须先于 `this` 的使用：

```
class Foo {}

class Bar extends Foo {
    constructor(num) {
        let tmp = num * 2; // OK
        this.num = num; // ReferenceError
        super();
        this.num = num; // OK
    }
}
```

在继承的构造器中不调用 `super()` 也会造成错误：

```
class Foo {}

class Bar extends Foo {
    constructor() {
    }
}

let bar = new Bar(); // ReferenceError
```

15.2.3.3 覆盖构造器的结果

就像在 ES5 中一样，可以通过显示地返回一个对象来覆盖构造器的结果：

```
class Foo {
    constructor() {
        return Object.create(null);
    }
}
console.log(new Foo() instanceof Foo); // false
```

如果这样做了，那么 `this` 是否已经被初始化都没关系了。换句话说，如果用这种方式覆盖结果，那么在继承的构造函数中没必要调用 `super()` 方法了。

15.2.3.4 类的默认构造器

如果没有为基类指定`构造器`，那么会使用下面的定义：

```
constructor() {}
```

对于继承类，会使用下面的默认构造器：

```
constructor(...args) {
    super(...args);
}
```

15.2.3.5 子类化内置构造器

在 ECMAScript 6 中，可以继承所有内置的构造器（ES5 中有相应的变通手段，但是有值得注意的限制）。

例如，现在可以创建自己的异常类（在大多数引擎中会继承堆栈特性）：

```
class MyError extends Error {
}
throw new MyError('Something happened!');
```

也可以创建 `Array` 的子类，该子类也会正确地处理 `length` ：

```
class MyArray extends Array {
    constructor(len) {
        super(len);
    }
}

// Instances of of `MyArray` work like real Arrays:
let myArr = new MyArray(0);
console.log(myArr.length); // 0
myArr[0] = 'foo';
console.log(myArr.length); // 1
```

注意，子类化内置构造器需要引擎本地化的支持，不能从转换器中得到这个特性。

15.3 类的细节

我们到目前为止看到的是类的概要。如果对类的背后原理感兴趣，那就继续阅读下去。让我们以类的语法开始，下面是一个来自于《Sect. A.4 of the ECMAScript 6 specification》的经过稍微修改的版本。

```
ClassDeclaration:
    "class" BindingIdentifier ClassTail
ClassExpression:
    "class" BindingIdentifier? ClassTail

ClassTail:
    ClassHeritage? "{" ClassBody? "}"
ClassHeritage:
    "extends" AssignmentExpression
ClassBody:
    ClassElement+
ClassElement:
    MethodDefinition
    "static" MethodDefinition
    ";"

MethodDefinition:
    PropName "(" FormalParams ")" "{" FuncBody "}"
    "*" PropName "(" FormalParams ")" "{" GeneratorBody "}"
    "get" PropName "(" ")" "{" FuncBody "}"
    "set" PropName "(" PropSetParams ")" "{" FuncBody "}"

PropertyName:
    LiteralPropertyName
    ComputedPropertyName
LiteralPropertyName:
    IdentifierName  /* foo */
    StringLiteral   /* "foo" */
    NumericLiteral  /* 123.45, 0xFF */
ComputedPropertyName:
    "[" Expression "]"
```

观察之后，得出两个结论：

- 被继承的值可以产生于任何表达式，这意味着可以写出如下的代码：

```
class Foo extends combine(MyMixin, MySuperClass) {}
```

- 允许方法之间有分号。

15.3.1 各种检查

- 错误检查：类名不能是 `eval` 或 `arguments` ；类元素名字不允许重复；名字 `constructor` 只能作为普通的方法名，不能用作 getter 、 setter 或者生成器方法。
- 类不能像函数一样调用。如果像函数一样调用，则会抛出 `TypeException` 异常。
- 原型方法不能用作构造器：

```
class C {
    m() {}
}
new C.prototype.m(); // TypeError
```

15.3.2 属性的特性（ Attributes of properties ）

Class declarations create (mutable) let bindings. 对于一个给定的类 `Foo` ：

- 静态的方法 `Foo.*` 是可写的和可配置的，但不可枚举。使它们可写就允许动态赋值。
- 一个构造器和它的属性 `prototype` 对象有一个不变的连接：
    - `Foo.prototype` 是不可写的，不可枚举的，不可配置的。
    - `Foo.prototype.constructor` 是不可写的，不可枚举的，不可配置的。
- 原型方法 `Foo.prototype.*` 是可写的和可配置的，但是不可枚举。

注意在对象字面量中的方法定义会产生可枚举的属性。

15.4 子类的细节

在 ECMAScript 6 中，子类看起来像下面这样。

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    ...
}

class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    ...
}

let cp = new ColorPoint(25, 8, 'green');
```

这段代码产生下面的对象。



下一小节讲解原型链（上图中的两列），该节的后面一节介绍了 `cp` 是如何分配内存和初始化的。

15.4.1 原型链

上上图中，可以看到有两条原型链（对象通过 `[[Prototype]]` 关系连接，该关系就是继承关系）：

- 左侧的一列：类（函数）。继承类的原型是它继承的类。基类的原型是 `Function.prototype` ， `Function.prototype` 也是函数的原型：

```
> const getProto = Object.getPrototypeOf.bind(Object);

> getProto(Point) === Function.prototype
true
> getProto(function () {}) === Function.prototype
true
```

- 右侧的一列：实例的原型链。一个类的整个目的就是设置这条原型链。这条原型链以 `Object.prototype` （ `Object` 的原型是 `null` ）结束，同

时，`Object.prototype` 也是通过对象字面量创建的对象的原型：

```
> const getProto = Object.getPrototypeOf.bind(Object);

> getProto(Point.prototype) === Object.prototype
true
> getProto({}) === Object.prototype
true
```

图中左侧一列表明了静态方法也会被继承。

15.4.2 分配和初始化实例

类构造器之间的数据流和 ES5 规范中的继承方法有差异。实际上，看起来大致如下所示：

```
// Instance is allocated here
function Point(x, y) {
    // Performed before entering this constructor:
    this = Object.create(new.target.prototype);

    this.x = x;
    this.y = y;
}
...

function ColorPoint(x, y, color) {
    // Performed before entering this constructor:
    this = uninitialized;

    this = Reflect.construct(Point, [x, y], new.target); // (A)
        // super(x, y);

    this.color = color;
}
Object.setPrototypeOf(ColorPoint, Point);
...

let cp = Reflect.construct( // (B)
             ColorPoint, [25, 8, 'green'],
             ColorPoint);
    // let cp = new ColorPoint(25, 8, 'green');
```

在 ES6 和 ES5 中，实例对象创建的地方不一样：

- 在 ES6 中，在基类的构造器中创建，构造器调用链的末端。
- 在 ES5 中，在 `new` 操作符中创建，构造器调用链的始端。

前面的代码使用了两个新的 ES6 特性：

- `new.target` 是一个隐式的参数，所有的函数都有。它用于构造器调用，而 `this` 用于方法调用。
  - 如果构造器已经显示地通过 `new` 调用了，它的值就是当前构造器（行 B）。
  - 如果构造器通过 `super()` 调用，它的值就是发出调用的构造器中的 `new.target` （行 A）。
  - 在一个普通的函数调用中，它的值是 `undefined` 。这意味着可以使用 `new.target` 来区分一个函数是否被当成普通函数调用还是被当成构造器调用（通过 `new` ）。
  - 在箭头函数内部， `new.target` 指向祖先作用域中最近的一个非箭头函数中的 `new.target` 。
- `Reflect.construct()` 完成构造器调用，最后一个参数用于指定 `new.target` 。

此种子类化方式的优点就是让普通的代码能够子类化内置的构造器（比如 `Error` 和 `Array` ）。后面一节讲解了为什么需要一种不同的方式。

15.4.2.1 安全检查

- 在继承的构造器中，如果在 `super()` 调用之前就访问 `this` ，会抛出错误，因为此时实例尚未创建， `this` 尚未初始化。
- 一旦 `this` 初始化了，调用 `super()` 就会产生一个 `ReferenceError` 错误。这避免了调用两次 `super()` 的问题。
- 如果构造器隐式返回（不使用 `return` ），那么返回结果就是 `this` 。如果 `this` 未被初始化，会抛出 `ReferenceError` 错误。这避免了忘记调用 `super()` 的问题。
- 如果构造器显示地返回一个非对象（包括 `undefined` 和 `null` ），最终返回结果就是 `this` （这种行为需要保持与 ES5 和更早的版本兼容）。如果 `this` 未被初始化，就会抛出 `TypeError` 错误。
- 如果构造器显示地返回一个对象，该对象会被用作最终返回结果。此时 `this` 是否初始化已经没有关系了。

15.4.2.2 `extends` 子句

让我们看看 `extends` 子句是如何影响类的设置的（ Sect. 14.5.14 of the spec ）。

`extends` 子句的值必须是“可构造的（ constructible ）”（可通过 `new` 调用）。但是允许为 `null` 。

```
class C {
}
```

- 构造器类型：基类构造器
- `C` 的原型： `Function.prototype` （类似于普通的函数）

- `C.prototype` 的原型： `Object.prototype` （也是通过对象字面量创建的对象的原型）

```
class C extends B {
}
```

- 构造器类型：继承造器
- `C` 的原型： `B`
- `C.prototype` 的原型： `B.prototype`

```
class C extends Object {
}
```

- 构造器类型：继承构造器
- `C` 的原型： `Object`
- `C.prototype` 的原型： `Object.prototype`

注意下面的代码与第一种情况的微妙区别：如果没有 `extends` 子句，那么类就是积累并且负责创建实例。如果一个类继承自 `Object` ，就是一个继承类，并且 `Object` 创建实例。最终的实例（包括它们的原型）都是一样的，但是实例构建过程是不一样的。

```
class C extends null {
}
```

- 构造器类型：继承构造器
- `C` 的原型： `Function.prototype`
- `C.prototype` 的原型： `null`

这种类就避免了 `Object.prototype` 出现在原型链中。但是基本没有什么用。并且，必须得小心：通过 `new` 调用这样的类会抛出错误，因为默认的构造器会调用父构造器，而 `Function.prototype` （父构造器）不能作为构造器调用。唯一一种避免该错的方式是添加一个返回对象的 `构造器` ：

```
class C extends null {
    constructor() {
        let _this = Object.create(new.target.prototype);
        return _this;
    }
}
```

`new.target` 确保 `C` 能正确地继承 - `_this` 的原型总是 `new` 的操作数。

15.4.3 为什么在 ES5 中不能子类化内置构造器？

在 ECMAScript 5 中，绝大多数构造器不能被子类化（有几种迂回方法）。

要理解为什么，让我们使用标准的 ES5 方式去子类化 `Array` 。我们将会很快看到，这是不行的。

```
function MyArray(len) {
    Array.call(this, len); // (A)
}
MyArray.prototype = Object.create(Array.prototype);
```

很不幸，如果实例化 `MyArray` ，我们发现不会正确地工作：在添加元素的时候，实例的 `length` 属性并不会自动变化：

```
> var myArr = new MyArray(0);
> myArr.length
0
> myArr[0] = 'foo';
> myArr.length
0
```

有两个障碍导致 `myArr` 无法成为一个正确的数组。

第一个障碍：初始化。传递给构造器 `Array` （行 A 处）的 `this` 会被忽略。这意味着不能使用 `Array` 来设置 `MyArray` 创建的实例。

```
> var a = [];
> var b = Array.call(a, 3);
> a !== b  // a is ignored, b is a new object
true
> b.length // set up correctly
3
> a.length // unchanged
0
```

第二个障碍：分配。通过 `Array` 创建的实例对象是特异的（ exotic ）（ ECMAScript 规范使用这个术语来描述那些有不同于普通对象特性的对象）： `Array` 创建的实例的属性 `length` 追踪和影响数组元素的管理。一般地，可以创建特异对象，但是不能将一个已存在的普通对象转换成特意对象。不幸的是，这就是 `Array` 必须要做的，在行 A 执行的时候：必须要将 `MyArray` 创建的普通的对象转换成特异对象。

15.4.3.1 解决方案： ES6 的子类化

在 ECMAScript 6 中，子类化 `Array` 看起来像下面这样：

```
class MyArray extends Array {
    constructor(len) {
        super(len);
    }
}
```

这会有效（但是转换器肯定不会支持，这依赖于 JavaScript 本地引擎支持）：

```
> let myArr = new MyArray(0);
> myArr.length
0
> myArr[0] = 'foo';
> myArr.length
1
```

现在可以看下 ES6 方式的子类化是怎么绕过这两个障碍的：

- 实例创建是在基构造器中进行的，这意味着 `Array` 可以生成一个特异的对象。相对于绝大多数的 `new` 一个对象的方式都依赖于子构造器的行为，这一步需要基构造器知道 `new.target` ，并且将 `new.target.prototype` 设置为分配出来的实例的原型。
- 实例初始化也会在基构造器中进行，派生的构造器获得初始化了的对象，并运用这个对象继续执行，而不是将自己的实例传递给父构造器，然后让父构造器去设置这个实例。

15.4.4 在方法中访问父属性

下面的 ES6 代码在行 B 调用了一个父方法：

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() { // (A)
        return `(${this.x}, ${this.y})`;
    }
}

class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    toString() {
        return super.toString() // (B)
            + ' in ' + this.color;
    }
}

let cp = new ColorPoint(25, 8, 'green');
console.log(cp.toString()); // (25, 8) in green
```

要理解父调用的原理，让我们看下 `cp` 的对象图：



`ColorPoint.prototype` 调用（行 B ）在本类中覆盖了的父类方法（开始于行 A ）。我们称方法存放的对象是此方法的宿主对象（ home object ）。例如， `ColorPoint.prototype` 是 `ColorPoint.prototype.toString()` 的宿主对象。

在行 B 调用父方法，总共包含三个步骤：

1、开始在宿主对象的原型链上搜索当前方法。2、找到名为 `toString` 的方法。3、用当前的 `this` 调用找到的方法。这样做的原因是：调用的父方法一定要能够访问到相同的实例属性（在我们的例子中，是 `cp` 的属性）。

注意，即便仅获取或设置一个属性（不是调用方法），仍然需要在第三步中考虑 `this` ，因为这个属性可能是通过 getter 或 setter 实现的。

让我们用三种不同的，但是等价的方式来描述这些步骤：

```
// Variation 1: super-method calls in ES5
var result = Point.prototype.toString.call(this) // steps 1,2,3

// Variation 2: ES5, refactored
var superObject = Point.prototype; // step 1
var superMethod = superObject.toString; // step 2
var result = superMethod.call(this) // step 3

// Variation 3: ES6
var homeObject = ColorPoint.prototype;
var superObject = Object.getPrototypeOf(homeObject); // step 1
var superMethod = superObject.toString; // step 2
var result = superMethod.call(this) // step 3
```

第三种就是 ECMAScript 6 处理父调用的方式。这种方式通过两个内部绑定的变量来支持，这两个绑定变量是函数内部环境提供的（函数环境给作用域中的变量提供存储空间，所谓的绑定变量）：

- [[thisValue]] ：该内部绑定变量在 ECMAScript 5 中也存在，存放在 `this` 值中。
- [[HomeObject]] ：指向环境函数的宿主对象。 [[HomeObject]] 是一个方法的内部属性，方法在被调用的时候， `super` 就会指向这个对象。绑定变量和内部属性都是 ECMAScript 6 新引入的。

> 现在方法是一种特殊的函数
>
> 在类中，一个使用 `super` 的方法定义会创建一种特殊的函数：仍然是一个函数，但是有内部的 `[[HomeObject]]` 属性。这个属性通过方法定义设定，在 JavaScript 代码中不能修改。因此，不可以自以为是地将这样的方法移到一个不同的对象上面去（但是这在将来的 ECMAScript 版本中可能可以这样做）。

15.4.4.1 什么地方可以使用 `super` ？

当原型链参与进来的时候，访问父属性就变得很方便，这就是为什么能在对象字面量和类（类既可以是被继承的，也可以是不被继承的；既可以是静态的，也可以是非静态的）的方法定义中使用 `super` 。

在下列场景中不能使用 `super` 访问属性：函数声明中，函数表达式中和生成器函数中。

15.4.4.2 不能移动使用 `super` 的方法

不能移动使用 `super` 的方法：这样的方法有一个内部的属性 `[[HomeObject]]` ，该属性与创建此方法的对象绑在一起。如果通过赋值的方式移动方法，方法中的 `[[HomeObject]]` 属性将继续指向原来的对象的父属性。在未来的 ECMAScript 版本中，可能会有一种方式来移动这样的方法。

15.5 种模式

在 ECMAScript 6 中，另一种使内置的构造器变得可扩展的机制：如果一个方法（比如 `Array.prototype.map()` 返回一个新实例），应该用什么构造器来创建这个实例呢？

在接下来的小节中会使用如下的辅助函数：

```
function isObject(value) {
    return (value !== null
       && (typeof value === 'object'
           || typeof value === 'function'));
}

/**
 * Spec-internal operation that determines whether `x` can be us
ed as a constructor.
 */
function isConstructor(x) {
    ...
}
```

15.5.1 标准的种模式

定制使用方法（比如 `Array.prototype.map()` ）创建的实例的模式被称为种模式：

- 如果存在 `this.constructor[Symbol.species]` ，使用它作为新实例的构造器。
- 否则，使用默认的构造器（例如数组的 `Array` ）。

用 JavaScript 代码实现，这种模式看起来像这样：

```
function SpeciesConstructor(O, defaultConstructor) {
    let C = O.constructor;
    if (C === undefined) {
        return defaultConstructor;
    }
    if (! isObject(C)) {
        throw new TypeError();
    }
    let S = C[Symbol.species];
    if (S === undefined || S === null) {
        return defaultConstructor;
    }
    if (! isConstructor(S)) {
        throw new TypeError();
    }
    return S;
}
```

> 数组的标准种模式在规范是通过 SpeciesConstructor() 实现的。

15.5.2 数组的种模式

下面代码大致描述了种模式是如何应用于数组的：

```
function ArraySpeciesCreate(originalArray, length) {
    let C = undefined;
    if (Array.isArray(originalArray)) {
        C = originalArray.constructor;
        if (isObject(C)) {
            C = C[Symbol.species];
        }
    }
    if (C === undefined || C === null) {
        return new Array(length);
    }
    if (! IsConstructor(C)) {
        throw new TypeError();
    }
    return new C(length);
}
```

`Array.prototype.map()` 返回的数组通过 `ArraySpeciesCreate(this, this.length)` 创建。

> 数组种模式在规范中是通过 ArraySpeciesCreate() 操作实现的。

15.5.3 静态方法中的种模式

Promise 中使用了大量的静态方法种模式，例如 Promise.all()：

```
let C = this; // default
if (! isObject(C)) {
    throw new TypeError();
}
// The default can be overridden via the property `C[Symbol.spec
ies]`
let S = C[Symbol.species];
if (S !== undefined && S !== null) {
    C = S;
}
if (!IsConstructor(C)) {
    throw new TypeError();
}
let instance = new C(···);
```

15.5.4 在子类中覆盖默认的 `spieces`

下面所示是 `[Symbol.species]` 的默认 getter ：

```
get [Symbol.species]() {
    return this;
}
```

默认的 `getter` 在内置的类 `Array` ， `ArrayBuffer` ， `Map` ， `Promise` ， `RegExp` ， `Set` 和 `%TypedArray%` 中都有实现，并且自动地被这些内置类的子类继承。

有两种方式可以覆盖默认的 `species` ：使用自定义的构造器或者使用 `null` 。

15.5.4.1 设置自定义构造器的 `species`

可以通过静态的 getter （行 A ）覆盖默认的 `species` ：

```
class MyArray1 extends Array {
    static get [Symbol.species]() { // (A)
        return Array;
    }
}
```

这样一来， `map()` 就返回 `Array` 的实例了：

```
let result1 = new MyArray1().map(x => x);
console.log(result1 instanceof Array); // true
```

如果不覆盖默认的 `species` ， `map()` 就会返回子类的实例：

```
class MyArray2 extends Array { }

let result2 = new MyArray2().map(x => x);
console.log(result2 instanceof MyArray2); // true
```

15.5.4.2 通过数据属性指定 `species`

如果不想使用静态的 getter ，那么就要使用 `Object.defineProperty()` 。你可以使用赋值，这样会触发 `setter` ，而这个 `setter` 并不存在（只有一个 getter ）。

例如，这里我们设置 `MyArray1` 的 `species` 为 `Array` ：

```
Object.defineProperty(
    MyArray1, Symbol.species, {
        value: Array
    });
```

15.5.4.3 将 `species` 设置为 `null`

如果将 `species` 设置为 `null` ，就会使用默认的构造器（选用哪个构造器决定于使用哪一个种模式变体，参考前面的小节获取更多相关信息）。

```
class MyArray3 extends Array {
    static get [Symbol.species]() {
        return null;
    }
}

let result3 = new MyArray3().map(x => x);
console.log(result3 instanceof Array); // true
```

15.6 有关类的常见问题解答

15.6.1 为什么类不能像函数一样调用？

目前禁止用函数的方式调用类。这样做是为将来做准备，为了最终通过类来添加一种处理函数调用的的方式。一种这样做的可能方式是通过一个特殊的方法（例如 `[Symbol.functionCall]` ）。

15.6.2 究竟如何实例化一个类，传递一组参数？

什么是 `Function.prototype.apply()` 的类模拟？那就是，如果有一个类 `TheClass` 和一组数组形式的参数 `args` ，如何实例化 `TheClass` ？

一种办法是通过扩展操作符（...）：

```
function instantiate(TheClass, args) {
    return new TheClass(...args);
}
```

另一种方法是使用 `Reflect.construct()` ：

```
function instantiate(TheClass, args) {
    return Reflect.construct(TheClass, args);
}
```

15.6.3 如何管理类的私有数据？

两种 ES5 的保持数据私有性的方式对类也有效：

- 1、可以让私有数据存放在类构造器的作用域环境中。我一直不喜欢因此而被迫添加特殊的实例方法。
- 2、可以使用命名约定（例如下划线前缀）来标记私有属性。我喜欢这种方式。这不会对私有属性提供完全的保护，并且略微使命名空间杂乱，但是实现的代码很简单。

在 ECMAScript 6 中，可以使用额外的 `WeakMaps` 来实现私有数据，这会对方法一中的私有数据提供完全地保护，但是代码更加优雅。下面是如何实现的代码，详细原理在关于 WeakMap 的那一章有讲解：

```
let _counter = new WeakMap();
let _action = new WeakMap();
class Countdown {
    constructor(counter, action) {
        _counter.set(this, counter);
        _action.set(this, action);
    }
    dec() {
        let counter = _counter.get(this);
        if (counter < 1) return;
        counter--;
        _counter.set(this, counter);
        if (counter === 0) {
            _action.get(this)();
        }
    }
}
```

15.6.4 下一个关于类的内容是什么？

类的设计原则是“最大程度地最小化”。讨论了几种高级的特性，但是最终被放弃了，为了得到一种 TC39 一致认同的设计。

下一个 ECMAScript 版本可以基于这个最小化的设计 - 类将会提供一些基础特性，比如 traits （或者 mixins ），值对象（如果两个不同的对象有相同的内容的话，那么它们是相等的）和常类（生成不可变实例）。

15.7 类的优点和缺点

在 JavaScript 社区里，类是有争议的：一方面，来自基于类的语言的人很开心，因为再也不需要应对 JavaScript 的非传统继承机制。另一方面，有很多 JavaScript 程序员认为 JavaScript 中复杂的不是原型继承，而是构造器。

ES6 的类提供了几点明显的好处：

- 兼容当前大量的代码。
- 相对于构造器和构造器继承，类使初学者更容易入门。
- 子类化在语言层面支持。
- 可以子类化内置的构造器。
- 不再需要继承库；框架之间的代码变得更加轻便。
- 为将来的高级特性奠定了基础： traits （或者 mixins ）， 不可变实例，等等。
- 使工具能够静态分析代码（ IDE ，类型检测器，代码风格检测器，等等）。

看几条关于 ES6 类的抱怨。你将会看到我赞同大多数抱怨，但是我同时认为利远大于弊。我很高兴 ES6 引入了类，并且推荐使用类。

15.7.1 抱怨： ES6 类掩盖了 JavaScript 继承的本质

是的， ES6 类确实掩盖了 JavaScript 继承的本质。类看起来的样子（语法上）和类的行为（类的语义）是没有联系的：看起来像是对象，但是是一个函数。对于类，我倾向于构造器对象，而不是构造器函数。我在 Proto.js 项目 中探索了这种方式，使用了一个小型的库（证明了使用这种方法是很有好处的）。

然而，向后兼容就有问题了，这就是为什么构造器函数也是有道理的原因。这种方式下， ES6 代码和 ES5 代码更容易相互操作。

语法和语义的不一致将会导致 ES6 和后续版本有一些阻力。但是可以简单地使用 ES6 表层的东西，由此来享受一种写代码的舒适感。我不认为这种舒适感在将来会拖累你。新手能够更快入门，然后才去学习背后的原理（在他们对语言感到更加自然之后）。

15.7.2 抱怨：类会禁锢你，因为强制性的 `new`

如果想实例化一个类，在 ES6 中必须要使用 `new` 。这意味着不能将类转换成工厂函数而不改变调用方式。确实是一个限制，但是有两个缓和的因素：

- 你可以覆盖 new 操作符的默认结果，从类的构造器方法中返回一个对象。
- 由于内置模块和类， ES6 使 IDE 更加容易地重构代码。因此，从新手到函数调用将会很简单。很明显，如果不控制调用你代码的代码（就像库的使用一样），这是不会对你有帮助的。

因此，类的确在语义上限制了你，但是，一旦 JavaScript 有 trait 了，类将不会在概念上限制你（关于面向对象的设计）。

15.8 深入阅读

下面的文档是本章的一个重要来源：

- 《 Instantiation Reform: One last time 》， Allen Wirfs-Brock 的幻灯片。
- [Speaking JS]，《 Keeping Data Private 》。

16 模块

本章讲解了 ES6 内置模块是如何工作的。

16.1 概览

在 ECMAScript 6 里，模块存放在文件中。文件和模块是严格一对一关系的。有两种从模块中导出东西的方式。两种方式可以混合使用，但是通常情况下分开使用比较好。

16.1.1 命名导出（ Named exports ）

可以有多个命名导出项：

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

可以引入整个模块：

```
//------ main.js ------
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

16.1.2 默认导出

可以只有单个默认导出。例如，一个函数：

```
//------ myFunc.js ------
export default function () { ··· } // no semicolon!

//------ main1.js ------
import myFunc from 'myFunc';
myFunc();
```

或者一个类：

```
//------ MyClass.js ------
export default class { ··· } // no semicolon!

//------ main2.js ------
import MyClass from 'MyClass';
let inst = new MyClass();
```

注意，如果默认导出一个函数或一个类的话，语句末尾是没有分号的（这在语义上叫做匿名声明（ anonymous declarations ））。

16.1.3 浏览器：脚本对比模块

	script	模块
HTML 元素	<script>	<script type="module">
顶级的变量是	全局变量（ global ）	模块的本地变量
顶级的 `this` 值	window	undefined
执行	同步地	异步地
声明式地引入（引入语句）	否	是
编程式地引入（基于 Promise 的 API ）	是	是
文件扩展名	.js	.js

16.2 JavaScript 中的模块

虽然 JavaScript 从来没有内置的模块，但是社区聚集于一种简单的模块风格，此模块风格在 ES5 和更早版本中通过库来实现。在 ES6 中也同样接受这种风格：

- 每一个模块都是一块代码，在加载完成后执行。
- 在这段代码中，可能有各种声明（变量声明，函数声明等等）。
    - 默认情况下，这些声明只是模块本地的。
    - 你可以把其中一些标记为导出，这样其它模块就可以引入了。
- 一个模块不仅能够导出东西，也可以从其它模块中导入东西。可以通过模块标识符或者字符串访问那些模块：
    - 相对路径（‘ `../model/user` ’）：这些路径相对于当前模块。通常文件扩展名 `.js` 可以省略。
    - 绝对路径（‘ `/lib/js/helpers` ’）：直接指向要引入的模块文件。
    - 名字（‘ `utils` ’）：需要配置模块名和所指向模块的映射关系。
- 模块是单例的。即使一个模块被引入了多次，仅会存在该模块的单一“实例”。
- 一个客户端或服务器应用的执行开始于一个初始化模块。在一些模块系统里面（包括 ES6 ），初始化模块可以内嵌在 HTML 文件中（考虑 <script> 元素）。

这种模块形式避免了全局变量，唯一全局的东西就是模块标识符。

16.2.1 ES5 模块系统

在语言层面上没有显示支持的前提下， ES5 模块系统的良好运作令人印象深刻。两个重要的标准（很不幸两者不兼容）是：

- **CommonJS** 模块 ：该规范主要实现于 Node.js 中（ Node.js 模块有一些特性超越了 CommonJS ）。特点：
    - 紧凑的语法
    - 为同步加载设计
    - 主要用于服务器端
- 异步的模块定义（ **AMD** ）：该标准最流行的实现是 RequireJS。特点：
    - 稍微复杂的语法，使 AMD 能够不借助于 eval() （或者说是一个编译步骤）。
    - 为异步加载设计
    - 主要用于浏览器端

上面是目前情况的一个简单说明。如果需要更深入的资料，可以参考 Addy Osmani 的 《Writing Modular JavaScript With AMD, CommonJS & ES Harmony》。

16.2.2 ECMAScript 6 模块

ECMAScript 6 模块的目标是创建一种模块化方式，让 CommonJS 使用者和 AMD 使用者都乐于接受：

* 类似于 CommonJS ，语法紧凑，倾向于导出单个内容，支持循环依赖。
* 类似于 AMD ，直接支持异步加载和可配置模块加载。

通过语言层面上的支持， ES6 模块超越了 CommonJS 和 AMD （后面详细解释）：

* 语法比 CommonJS 更加紧凑。
* 可以静态分析代码结构（比如静态检测，优化，等等）。
* 比 CommonJS 更好地支持循环依赖。

ES6 模块标准有两个部分：

* 声明式的语法（在引入和导出的时候）
* 可编程的加载器 API ：可以配置模块如何加载，并且可以按条件加载模块

16.3 ES6 模块初窥

有两种导出的方式：命名导出（每个模块可以导出若干个）和默认导出（每个模块只能导出一个）。

16.3.1 命名导出（每个模块可以导出若干个）

通过在声明的前面加上 `export` 关键字，一个模块可以导出多个内容。这些导出的内容通过名字区分，被称为命名导出。

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

还有其它方式指定命名导出（后面讲解），但是我发现这种方式非常方便：尽情书写代码，就像没有外部的世界一样，然后给想要导出的所有内容加上一个关键字标签。

也可以引入整个模块，然后通过对象属性的方式来访问命名导出的内容：

```
//------ main.js ------
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

在 **CommonJS** 中有相似的语法： 有段时间，在 Node.js 中，我尝试几种看起来很聪明的方法去处理模块导出，以便少写一些多余的代码。现在我倾向于下面这种简单的但是稍微啰嗦的风格，这不禁让人联想到文章《 revealing module pattern 》：

```
//------ lib.js ------
var sqrt = Math.sqrt;
function square(x) {
    return x * x;
}
function diag(x, y) {
    return sqrt(square(x) + square(y));
}
module.exports = {
    sqrt: sqrt,
    square: square,
    diag: diag,
};

//------ main.js ------
var square = require('lib').square;
var diag = require('lib').diag;
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

16.3.2 默认导出（每个模块只能导出一个）

仅导出单一值的模块在 Node.js 社区中非常流行。但在前端开发中也很常见，经常为 model 写一些构造器或者类，每个模块一个 model 。一个 ECMAScript 6 模块可以只有一个默认导出。默认导出的内容特别方便引入。

下面的 ECMAScript 6 模块“是”一个单一的函数：

```
//------ myFunc.js ------
export default function () { ··· } // no semicolon!

//------ main1.js ------
import myFunc from 'myFunc';
myFunc();
```

默认值是类的 ECMAScript 6 模块看起来像下面这样：

```
//------ MyClass.js ------
export default class { ··· } // no semicolon!

//------ main2.js ------
import MyClass from 'MyClass';
let inst = new MyClass();
```

16.3.2.1 `export default` 的操作数：声明或表达式

在语法上， `export default` 的操作数是：

- 一个声明：如果以函数或类开始。这意味着在前面的代码示例中，是匿名声明（仅能用于此种特殊情形）。
- 一个表达式：所有其余场景。

因此，有两种可以默认导出匿名函数的方法：

```
export default function () {} // function declaration
export default (function () {}); // function expression
```

也可以给函数声明一个名字。如果想在模块的其它位置访问默认导出的内容，这会很有用：

```
bar();

export default function bar() { ··· }
```

这段代码等价于：

```
bar();

function bar() { ··· }

export default bar;
```

16.3.3 模块导出的是不变的绑定，而不是值

相对于 AMD 模块和 CommonJS 模块， ES6 模块并不导出值（值的快照），而是绑定（指向值存储位置的引用）。这些绑定是不可变的：只能通过绑定读取值，不能修改值。但是可以从模块内部修改。下面的代码说明了这是如何运作的：

```
//------ lib.js ------
export let mutableValue = 3;
export let incMutableValue() {
    mutableValue++;
}

//------ main1.js ------
import { mutableValue, incMutableValue } from './lib';

// The imported value is live
console.log(mutableValue); // 3
incMutableValue();
console.log(mutableValue); // 4

// The imported value can’t be changed
mutableValue++; // TypeError
```

如果通过星号（*）引入，会得到类似的结果：

```
//------ main2.js ------
import * as lib from './lib';

// The imported value is live
console.log(lib.mutableValue); // 3
lib.incMutableValue();
console.log(lib.mutableValue); // 4

// The imported value can’t be changed
lib.mutableValue++; // TypeError
```

16.3.4 引入的内容会提升

模块引入会提升（在内部处理中，会将引入移到当前作用域的最前面）。因此，在一个模块中，何处引入是没有关系的，下面的代码也并没有什么问题：

```
foo();

import { foo } from 'my_module';
```

16.3.5 模块文件就是普通的 JavaScript 文件

下面所有种类的文件都有相同的文件扩展名 `.js` ：

- 1、 ECMAScript 6 模块
- 2、 CommonJS 模块
- 3、 AMD 模块

- 4、脚本文件（在 HTML 文件中通过 <script src="file.js"> 加载）

也就是说，所列出的所有文件都是“ JavaScript 文件”。它们被如何解析，依赖于所在的使用环境。

16.4 设计目标

16.5 更多关于引入和导出的知识

16.5.1 引入代码的风格

ECMAScript 6 几种引入的代码风格：

- 默认引入：

```
import localName from 'src/my_lib';
```

- 命名空间方式引入：将模块作为一个对象引入（每一个属性指代一个命名导出）。

```
import * as my_lib from 'src/my_lib';
```

- 命名引入

```
import { name1, name2 } from 'src/my_lib';
```

可以对命名引入进行重命名：

```
// Renaming: import `name1` as `localName1`
import { name1 as localName1, name2 } from 'src/my_lib';
```

- 空引入：仅加载模块，并不引入任何内容。在程序中第一次这种引入会执行模块内的代码。

```
import 'src/my_lib';
```

仅有两种方法组合使用上述风格，并且它们出现的顺序是固定的；默认导出总是放在第一个位置。

- 组合默认引入和命名空间引入：

```
import theDefault, * as my_lib from 'src/my_lib';
```

组合默认引入和命名引入：

```
import theDefault, { name1, name2 } from 'src/my_lib';
```

16.5.2 导出风格：内联和子句

有两种方式可以导出当前模块中的东西。一种是，可以用 `export` 关键字标记声明。

```
export var myVar1 = ···;
export let myVar2 = ···;
export const MY_CONST = ···;

export function myFunc() {
    ...
}
export function* myGeneratorFunc() {
    ...
}
export class MyClass {
    ...
}
```

默认导出的“操作数”是一个表达式（包括函数表达式和类表达式）。例如：

```
export default 123;
export default function (x) {
    return x
}
export default x => x;
export default class {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

另一种，可以在模块结束处列出所有想导出的东西。

```
const MY_CONST = ···;
function myFunc() {
    ...
}

export { MY_CONST, myFunc };
```

也可以用不同的名字导出：

```
export { MY_CONST as FOO, myFunc };
```

16.5.3 重导出（ re-exporting ）

重导出指的是把另一个模块的导出添加到当前模块的导出中去。可以添加其它模块所有的导出：

```
export * from 'src/other_module';
```

`export *` 会忽略默认导出。

也可以选择性地导出（同时可以重命名）：

```
export { foo, bar } from 'src/other_module';

// Renaming: export other_module's foo as myFoo
export { foo as myFoo, bar } from 'src/other_module';
```

16.5.3.1 使重导出成为默认导出

下面的语句是另一个模块 `foo` 的默认导出成为当前模块的默认导出：

```
export { default } from 'foo';
```

下面的语句使模块 `foo` 的命名导出 `myFunc` 成为当前模块的默认导出：

```
export { myFunc as default } from 'foo';
```

16.5.4 所有导出风格

ECMAScript 6 提供了几种导出风格：

- 重导出：
    - 重导出所有东西（除了默认导出）：

      ```
      export * from 'src/other_module';
      ```

    - 通过子句重导出：

      ```
      export { foo as myFoo, bar } from 'src/other_module';
      ```

- 通过子句重导出

```
export { MY_CONST as FOO, myFunc };
```

- 内联导出：
  - 变量声明：

    ```
    export var foo;
    export let foo;
    export const foo;
    ```

  - 函数声明：

    ```
    export function myFunc() {}
    export function* myGenFunc() {}
    ```

  - 类声明：

    ```
    export class MyClass() {}
    ```

- 默认导出：
  - 函数声明（可以是匿名的，但仅限于此处）：

    ```
    export default function myFunc() {}
    export default function () {}

    export default function* myGenFunc() {}
    export default function* () {}
    ```

  - 类声明（可以使匿名的，但仅限于此处）：

    ```
    export default class MyClass() {}
    export default class () {}
    ```

  - 表达式：导出值，而不是绑定。注意结束的分号。

    ```
    export default foo;
    export default 'Hello world!';
    export default 3 * 7;
    export default (function () {});
    ```

16.5.5 在一个模块中同时包含命名导出和默认导出

下面的场景在 JavaScript 中出人意料地常见：一个库只是一个单一的函数，但是附加的功能通过此函数的属性来提供。例如 jQuery 和 Underscore.js 。下面是 Underscore 作为一个 CommonJS 模块的粗略描绘：

```
//------ underscore.js ------
var _ = function (obj) {
    ...
};
var each = _.each = _.forEach =
    function (obj, iterator, context) {
        ...
    };
module.exports = _;

//------ main.js ------
var _ = require('underscore');
var each = _.each;
```

有了 ES6 ，函数 `_` 就是默认导出， `each` 和 `forEach` 是命名导出。事实证明，可以同时使用命名导出和默认导出。在下例中，前面的 CommonJS 模块，被重写为 ES6 模块，看起来像这样：

```
//------ underscore.js ------
export default function (obj) {
    ...
}
export function each(obj, iterator, context) {
    ...
}
export { each as forEach };

//------ main.js ------
import _, { each } from 'underscore';
...
```

注意， CommonJS 的版本和 ECMAScript 6 的版本大致相似。后者有一个平整的解构，前者是内嵌的。

对于 ES6 模块，我通常推荐在一个单一的模块中，要么仅有默认导出，要么仅有命名导出。也就是说，不应该混合这两种导出风格。

16.5.5.1 默认导出只是另一种命名导出而已

默认导出实际上只是拥有特殊名字 `default` 的命名导出。也就是说，下面两个语句是等价的：

```
import { default as foo } from 'lib';
import foo from 'lib';
```

类似地，下面的两个模块有相同的默认导出：

```
//------ module1.js ------
export default function foo() {} // function declaration!

//------ module2.js ------
function foo() {}
export { foo as default };
```

16.5.5.2 `default` ：可以作为导出名，但是不能作为变量名

不能使用保留字（比如 `default` 和 `new` ）作为变量名，但是可以用于导出的名字（在 ECMAScript 5 中也可以用作属性名）。如果想直接引入这样的命名导出，就必须将其重命名为正确的变量名。

这意味着 `default` 只能在重命名引入的左侧出现：

```
import { default as foo } from 'some_module';
```

只能在重命名导出的右侧出现：

```
export { foo as default };
```

在重导出中， `as` 左右两侧都是导出名：

```
// The following two statements are equivalent:
export { default } from 'foo';
export { default as default } from 'foo';

export { myFunc as default } from 'foo';
export { default as otherFunc } from 'foo';
```

16.6 ECMAScript 6 模块加载器 API

模块除了声明式的语法之外，也有可动态编程的 API 。这允许你：

- 通过动态编程方法使用模块
- 配置模块加载

> **模块加载器 API 不是 ES6 标准的组成部分**
>
> 将会在另外一份文档中说明，“ JavaScript 加载器标准”，此标准会比语言规范更加充满活力地改进。该文档的仓库表明：
>
> [JavaScript 加载器规范] 巩固了 ECMAScript 模块加载语义化的工作，同时满足了 Web 浏览器和 Node.js 的模块加载需求。
>
> **模块加载器 API 仍然处于制定过程中**
>
> 正如你在《 the repository of the JavaScript Loader Standard 》中看到的一样，模块加载器 API 仍然处于制定过程中。所有你在本书中读到的相关内容都是试探性的。要得到一个关于这些 API 样子的初步印象，可以看一下 GitHub 上的《the ES6 Module Loader Polyfill》 。

16.6.1 加载器

加载器负责解析模块标识符（在 `import-from` 后的字符串 ID ），加载模块，等等。它们的构造器是 Reflect.Loader 。每一个平台都在全局变量 `System` （系统加载器）中维护一个默认实例，该实例实现了模块加载的特殊风格。

16.6.2 加载器方法：引入模块

可以通过动态编程的方式引入模块，使用基于 Promise 的 API ：

```
System.import('some_module')
.then(some_module => {
    // Use some_module
})
.catch(error => {
    ...
});
```

`System.import()` 使你能够：

- 在 <script> 元素中（此处不支持模块语法，详细内容参考关于模块和脚本对比的小节）使用模块。
- 根据条件加载模块。

`System.import()` 获取一个单一的模块，可以使用 `Promise.all()` 来引入若干个模块：

```
Promise.all(
    ['module1', 'module2', 'module3']
    .map(x => System.import(x)))
.then(([module1, module2, module3]) => {
    // Use module1, module2, module3
});
```

16.6.3 更多加载器方法

加载器还有很多方法，其中有三个重要的：

- `System.modules(source, options?)`
  新建一个模块实例，并在其中执行 `source` 指定的 JavaScript 代码，然后通过 Promise 异步返回创建的模块实例。
- `System.set(name,module)`
  用于注册一个模块（例如通过 `System.module()` 创建的）。
- `System.define(name,source,options?)`
  执行 `source` 中的模块代码，然后注册返回的模块实例。

16.6.4 配置模块加载

模块加载器 API 将会有大量的钩子用于配置加载过程。使用场景包括：

- 1、在模块引入的时候检测模块是否有语法错误（例如，通过 JSLint 或 JSHint）。
- 2、在引入的时候自动转换模块（有的模块可能包含 CoffeeScript 或者 TypeScript 代码）。
- 3、使用遗留的模块（ AMD ， Node.js ）。

可配置的模块加载在 Node.js 和 CommonJS 中是做不到的。

16.7 在浏览器中使用 ES6 模块

让我们看一下浏览器是如何支持 ES6 模块的。

> 浏览器对于 **ES6** 模块的支持工作正在进行
>
> 类似于模块加载，浏览器中对于模块其它方面的支持也正在进行中。所有你在此处读到的内容都可能改变。

16.7.1 浏览器：异步模块对比同步脚本

在浏览器中，有两种不同的实体：脚本和模块。它们有略微不同的语法，行为也不一样。

下面是一个不同点的概览，后面会详细讲述：

	脚本	模块
HTML 元素	<script>	<script type="module">
顶级的变量是	全局变量（global）	模块的本地变量
顶级的 `this` 值	window	undefined
执行	同步地	异步地
声明式地引入（引入语句）	否	是
编程式地引入（基于 Promise 的 API）	是	是
文件扩展名	.js	.js

16.7.1.1 Script

Script 是浏览器中传统的嵌入 JavaScript 脚本和引用外部 JavaScript 文件的方式。Script 有一个网络媒体类型属性，该属性用作：

- web 服务器传回来的 JavaScript 文件的内容类型。
- <script> 元素的 `type` 属性值。注意，对于 HTML5，如果 <script> 元素包含或者引用的是 JavaScript，则推荐省略 `type` 属性。

下面是最重要的取值：

- `text/javascript`：是一个遗留的值，如果省略 script 标签的 `type` 属性，默认值就是它。对于 Internet Explorer 8 和更早版本的浏览器来说，这是最安全的选择。

- `application/javascript` ：现代浏览器中推荐使用。

JavaScript 线程在代码加载并执行完之后停止。

16.7.1.2 模块

为了与 JavaScript 通常的运行完整性语义保持一致，模块体的执行不能被打断。这给导入模块留下了两个选择：

- 1、在模块体执行的过程当中，同步加载指定的外部模块。这就是 Node.js 的方式。
- 2、在模块体执行之前，异步地加载指定的所有模块。这就是 AMD 模块处理的方式，是浏览器环境的最佳选择，因为模块通过互联网加载，在模块加载的时候没有必要停止代码的执行。另一个好处是，这种方式允许并行地加载多个模块。

ECMAScript 6 提供了两种场景下都是最好的方式：Node.js 中的同步语法加上 AMD 的异步加载。 ES6 模块在语法上比 Node.js 模块简单：引入和导出必须要在顶级范围使用，这也意味着不能根据条件来引入。这种约束使 ES6 模块加载器能够静态分析当前模块引入了些什么模块，并在当前模块体执行之前加载好需要的模块。

脚本同步的本质使它们不能成为模块。脚本甚至不能声明式地引入模块（如果想引入模块，只能使用可动态编程编程引入的模块加载器 API ）。

在浏览器中，模块可以通过新的 <script> 元素形式使用，这种方式是完全异步的：

```
<script type="module">
    import $ from 'lib/jquery';
    var x = 123;

    // The current scope is not global
    console.log('$' in window); // false
    console.log('x' in window); // false

    // `this` still refers to the global object
    console.log(this === window); // true
</script>
```

正如你看见的一样， script 元素有自己的作用域，在里面的变量是该作用域的本地变量。注意，模块代码默认是处于严格模式下的。这是一个好消息 - 不用再写 `'use strict'` 了。

类似于普通的 <script> 元素， <script> 也可用于加载外部的模块。例如，下面的标签通过 `main` 模块启动 web 应用（ `import` 属性是我发明的，现在还不清楚会用什么名字）。

```
<script type="module" import="impl/main"></script>
```

在 HTML 中通过自定义 <script> 类型来支持模块的优点是：在老的浏览器中可以很容易地通过 polyfill （一种库）来引入这种支持。最终可能是也可能不是一个专用的模块元素（例如 `module` ）。

16.7.2 打包

现代的 web 应用由很多通常比较小的模块组成。通过 HTTP 加载这些模块会带来性能上的负面影响，因为每一个模块都需要一个独立的请求。因此，在 web 开发世界里，将多个模块放在一个文件中是一种运用很久的传统做法。当前的方法很复杂，容易出错，并且仅对 JavaScript 有效。有两种解决办法：

- HTTP/2 ：将允许在一个 TCP 连接中执行多个请求，这使打包显得不是那么必要了。然后可以增量地更新应用，因为如果某一个模块改变了，浏览器没必要重新下载整个打包文件。
- 包：另外， W3C 技术架构组正在制定一份用于“在 Web 环境上打包”的规范。想法是将整个文件夹放在一个包里面（想一下 ZIP 文件，但是是一种不同的格式）。然后这种包 URL ：

```
http://example.org/downloads/editor.pack#url=/root.html;fragment
=colophon
```

等价于下面这种普通的 URL ，如果这个包在服务器的根路径下打开的话：

```
http://example.org/root.html#colophon
```

浏览器必须确保一旦你在一个包里面，那么相对 URL 就要能按照预期工作。

> 本节来源
>
> - 《 Modules: Status Update 》， David Herman 的幻灯片。
> - 《 Modules vs Scripts 》，David Herman 的一封邮件。

16.8 关于模块的常见问题解答

16.8.1 命名导出是必须的吗？为什么不简单地导出对象？

你可能想知道 - 如果我们可以简单地默认导出对象（就像在 CommonJS 中那样），为什么还需要命名导出？答案就是你不能通过对象来强制形成一个静态的语法结构，随后便丧失掉所有相关的优点（在 16.4.2 节有讲解）。

16.8.2 我可以通过 `eval()` 执行模块体吗？

不行，不能这么做。对于 `eval()` 来说，模块是很高层次的构造器。模块加载器 API 提供了从字符串创建模块的方式。在语法上， `eval()` 接收脚本而不是模块。

16.9 ECMAScript 6 模块的益处

乍一看， ECMAScript 6 的内置模块可能看起来像是一个无聊的特性 - 毕竟，我们已经有几种不错的模块系统。但是 ECMAScript 6 模块有一些特性是库无法提供的，比如更加紧凑的语法和静态的模块结构（这对代码优化、静态检测等等有帮助）。同时也能够 - 充满希望地 - 结束当前占主导的标准 CommonJS 和 AMD 之间的分裂格局。

模块有一个单一的本地标准意味着：

- 不再有 UMD （ Universal Module Definition ）： UMD 是一种模式的名字，该模式允许同一个文件可以在若干个模块系统中使用（例如 CommonJS 和 AMD ）。一旦 ES6 成为唯一的模块标准， UMD 就会成为过去。
- 新的浏览器 API 会变得模块化，而不是全局变量或者 `navigator` 的属性。
- 不再用对象作为命名空间：像 `Math` 和 `JSON` 这种对象在 ECMAScript 5 中起着命名空间的作用。在将来，这种功能可以通过模块实现。

16.10 深入阅读

- **CommonJS** 对比 **ES6**：《JavaScript Modules》（作者 Yehuda Katz）是一个 ECMAScript 6 的快速介绍。第二页特别有趣，并排展示了 CommonJS 模块和 ECMAScript 6 版本的模块。

[规范] 创建绑定的函数：CreateImportBinding() [规范] 派别。“Imports”开始于语法规则，并继续维持语义。[规范] 规范中的方法 GetExportedNames() 搜集一个模块的导出内容。在规范中的第七步，会执行一个检查，防止其它模块的默认导出被重导出。[规范] 派别。Exports”开始于语法规则，并继续维持语义。

17 新的数组特性

本章阐述了 ES6 中数组的新特性。

17.1 新的数组类方法

Array 类对象上增加了几个专有方法。

17.1.1 Array.from(arrayLike, mapFunc?, thisArg?)

`Array.from()` 方法的基本功能是将下面两种类型的对象转换成数组：

- 类数组对象（Array-like objects），它具有 length 属性和索引元素，比如像 `document.getElementsByClassName()` 这种 DOM 操作的返回值。
- 可迭代的对象（Iterable object），它的内容可以每次被检索一个元素。数组是可迭代的，同样的 ES6 中新的数据结构，Map 和 Set 也是可迭代的。

下面的代码是将类数组对象（Array-like objects）转换成数组：

```
let lis = document.querySelectorAll('ul.fancy li');
Array.from(lis).forEach(function (li) {
    console.log(node);
});
```

`querySelectorAll()` 方法返回的结果不是一个数组，所以它没有 `forEach()` 方法，这也正是我们为什么要将它转换成数组的原因。

17.1.1.1 Mapping via Array.from()

一般地，对于使用 `map()` 方法的场景来说， `Array.from()` 也是一个便捷的方案：

```
let spans = document.querySelectorAll('span.name');

// map(), generically:
let names1 = Array.prototype.map.call(spans, s => s.textContent)
;

// Array.from():
let names2 = Array.from(spans, s => s.textContent);
```

在这个例子中， `document.querySelectorAll()` 返回的结果依然是一个类数组对象（Array-like object），而不是一个数组， 这就是为什么我们不能在它上面直接调用 `map()` 方法。在之前的例子，我们将类数组对象(Array-like object)转换成数组以调用 `forEach()` 方法。在这里，我们通过调用 `通用方法` 和双参数版本的 `Array.from()` 方法跳过这个中间步骤。

> 译者注：通用方法指的是：首先是一个方法，然后是这个方法能用于多种类型的对象。比如 `Array.prototype.slice()` 方法不仅仅可用于数组，也可用于类数组对象、字符串。

17.1.1.2 数组中的空缺(holes)

`Array.from()` 会忽略数组中的空缺，将它们当做 undefined 元素进行处理：

```
> Array.from([0,,2])
[ 0, undefined, 2 ]
```

这意味这也可以使用 `Array.form()` 方法创建和填充数组：

```
> Array.from(new Array(5), () => 'a')
[ 'a', 'a', 'a', 'a', 'a' ]

> Array.from(new Array(5), (x,i) => i)
[ 0, 1, 2, 3, 4 ]
```

如果你想使用固定的值填充数组（前面的两个例子中的第一个）那么 `Array.prototype.fill()` 方法（见下文）会是一个更好的选择。

17.1.1.3 from() 在 Array 子类中的使用

另一种 `Array.from()` 方法的使用情况是将类数组对象（Array-like object）或可迭代对象（Iterable object）转换成 Array 子类的一个实例，举例，如果你创建一个 Array 类的子类 MyArray，并且想将这样一个对象转换成 MyArray 的一个实例，你只需使用 `MyArray.from()` 。究其原因，是因为在 ES6 中构造函数之间的继承。（超类的构造函数是其子类构造函数的原型）

```
class MyArray extends Array {
    ...
}
let instanceOfMyArray = MyArray.from(anIterable);
```

你也可以将该功能和 map 一起使用，对你构造函数产生的结果进行一次 map 操作：

```
// from() – determine the result’s constructor via the receiver
// (in this case, MyArray)
let instanceOfMyArray = MyArray.from([1, 2, 3], x => x * x);

// map(): the result is always an instance of Array
let instanceOfArray   = [1, 2, 3].map(x => x * x);
```

17.1.2 Array.of(...items)

Array.of(item_0, item_1, ···) 可以创建一个由 item_0、item_1 等元素组成的数组。

17.1.2.1 Array.of() 可以作为 Array 字面量用于 Array 子类

如果你想把一些值转换成数组，你应当一直使用 Array 字面量方式，特别是当 Array 构造函数不能用的情况下，例如只有一个简单参数且该参数是数字的情况：

```
> new Array(3, 11, 8)
[ 3, 11, 8 ]
> new Array(3)
[ , ,  ,]
> new Array(3.1)
RangeError: Invalid array length
```

但是你如何把值转换成数组子类构造函数的实例？`Array.of()` 方法帮助你实现（记住，Array 子类的构造函数会继承所有的 Array 类的方法，包括 `of()` 方法）。

```
class MyArray extends Array {
    ...
}
console.log(MyArray.of(3, 11, 8) instanceof MyArray); // true
console.log(MyArray.of(3).length === 1); // true
```

17.2 新的数组原型方法

一些可以被用于数组实例的新方法

17.2.1 遍历数组

下面的方法可以帮助遍历数组：

```
Array.prototype.entries()
Array.prototype.keys()
Array.prototype.values()
```

上述各方法的返回的结果都是包含一组值的序列，但这些值并不是作为一个数组返回；它们通过迭代器被一个一个遍历出来。让我们来看一个例子。使用 `Array.from()` 把迭代器中的内容放入数组：

```
> Array.from(['a', 'b'].keys())
[ 0, 1 ]
> Array.from(['a', 'b'].values())
[ 'a', 'b' ]
> Array.from(['a', 'b'].entries())
[ [ 0, 'a' ],
  [ 1, 'b' ] ]
```

也可以使用展开操作符（...）将迭代器转换成数组：

```
> [...['a', 'b'].keys()]
[ 0, 1 ]
```

17.2.1.1 遍历 [index, element] 对

你可以结合 ECMAScript 6 的 `for-of` 循环与 `entries()` 方法去方便地遍历 `[index, element]` 对。

```
for (let [index, elem] of ['a', 'b'].entries()) {
    console.log(index, elem);
}
```

17.2.2 检索数组中的元素

`Array.prototype.find(predicate, thisArg?)`

返回数组中被回调函数判断为真的第一元素，如果没有该元素，则返回 undefined ，举例：

```
> [6, -5, 8].find(x => x < 0)
-5
> [6, 5, 8].find(x => x < 0)
undefined
```

`Array.prototype.findIndex(predicate, thisArg?)`

返回数组中被回调函数判断为真的第一个元素的索引，如果没有该元素，则返回-1。举例：

```
> [6, -5, 8].findIndex(x => x < 0)
1
> [6, 5, 8].findIndex(x => x < 0)
-1
```

回调函数的完整写法:

`predicate(element, index, array)`

17.2.2.1 findIndex() 方法将数组中的空缺（holes）作为 undefined 元素处理

`findIndex()` 方法将数组中的空缺（holes）作为 undefined 元素处理（使用 `find()` ，你将无法告诉它是否存在，因为如果它不能找到任何东西，它将返回 undefined ）：

```
> ['a',,'c'].findIndex(x => x === undefined)
1
```

17.2.2.2 通过 findIndex() 方法寻找 NaN

`Array.prototype.indexOf()` 的一个众所周知的局限性在于它无法找到NaN 的，因为通过===对元素进行搜索：

```
> [NaN].indexOf(NaN)
-1
```

与 `findIndex()` 方法一起，你可以使用 `Object.is()` 方法（在面向对象的章节解释），它没有这样的问题：

```
> [NaN].findIndex(y => Object.is(NaN, y))
0
```

你也可以采用更通用的方法，通过创建一个辅助函数的 `elemIs()` ：

```
> function elemIs(x) { return Object.is.bind(Object, x) }
> [NaN].findIndex(elemIs(NaN))
0
```

17.2.3 Array.prototype.fill(value, start?, end?)

用给定的值填充数组：

```
> ['a', 'b', 'c'].fill(7)
[ 7, 7, 7 ]
```

数组中的空缺（holes）不会被特殊处理：

```
> new Array(3).fill(7)
[ 7, 7, 7 ]
```

你可以限定填充范围（起始和结束）：

```
> ['a', 'b', 'c'].fill(7, 1, 2)
[ 'a', 7, 'c' ]
```

21 生成器（ Generator ）

生成器是 ECMAScript 6 的新特性，是可以暂停（ pause ）和唤醒（ resume ）的函数（考虑协同多任务处理或者协同程序）。它对很多应用程序都有帮助：迭代器，异步编程，等等。本章讲解了生成器是如何工作的，大致展示一下在具体应用中的使用。

> 下面的 GitHub 仓库包含了示例代码： generator-examples

21.1 概览

两个重要的生成器应用场景是：

- 实现迭代功能
- 阻塞异步函数调用

下面的子节简单介绍了两个应用场景，更详尽的内容在后面讲解（还会讨论其它一些场景）。

21.1.1 通过生成器实现迭代功能

下面的函数返回一个可迭代的对象，每个迭代元素都是一个 [key, value] 对：

```
// The asterisk after `function` means that
// `objectEntries` is a generator
function* objectEntries(obj) {
    let propKeys = Reflect.ownKeys(obj);

    for (let propKey of propKeys) {
        // `yield` returns a value and then pauses
        // the generator. Later, execution continues
        // where it was previously paused.
        yield [propKey, obj[propKey]];
    }
}
```

关于 `objectEntries()` 是如何工作的问题在后面讲解。它的使用就像这样：

```
let jane = { first: 'Jane', last: 'Doe' };
for (let [key,value] of objectEntries(jane)) {
    console.log(`${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe
```

21.1.2 阻塞异步函数调用

在下面的代码中，我使用流程控制库 co 来异步获取两个 JSON 文件。请注意，行 A 处的执行块直到 `Promise.all()` 的结果准备就绪之后才会执行。这意味着看起来像是同步的代码却执行了异步的操作。

```
co(function* () {
    try {
        let [croftStr, bondStr] = yield Promise.all([  // (A)
            getFile('http://localhost:8000/croft.json'),
            getFile('http://localhost:8000/bond.json'),
        ]);
        let croftJson = JSON.parse(croftStr);
        let bondJson = JSON.parse(bondStr);

        console.log(croftJson);
        console.log(bondJson);
    } catch (e) {
        console.log('Failure to read: ' + e);
    }
});
```

`getFile(url)` 获取 url 指向的文件，具体实现在后面展示，我也将会介绍 `co` 是如何工作的。

21.2 什么是生成器？

生成器就是可以暂停（pause）和唤醒（resume）的函数（考虑协同多任务处理和协同程序），这使得大量应用变为可能。

作为第一个例子，考虑下面名为 `genFunc` 的生成器函数：

```
function* genFunc() {
    console.log('First');
    yield; // (A)
    console.log('Second'); // (B)
}
```

`genFunc` 在两个方面和普通的函数声明不一样：

- 以“关键字” `function*` 开始。
- 在函数中的 yield 处暂停。

调用 `genFunc` 并不会执行它。相反，它返回一个所谓的生成器对象让我们能够控制 `genFunc` 的执行：

```
> let genObj = genFunc();
```

`genFunc()` 初始的时候在它的函数体开始处暂停。方法 `genObj.next()` 继续 `genFunc` 的执行，直到下一个 `yield` ：

```
> genObj.next()
First
{ value: undefined, done: false }
```

正如你在最后一行看到的， `genObj.next()` 也返回一个对象，现在先忽略它。一旦把生成器当做迭代器的时候，这就很重要了。

`genFunc` 现在暂停在行 A 处。如果我们再次调用 `next()` ，就会唤醒执行过程，行 B 得以执行：

```
> genObj.next()
Second
{ value: undefined, done: true }
```

然后，函数完成，执行流程离开函数体，继续调用 `genObj.next()` 不再有效果。

21.2.1 创建生成器的方式

有四种方法可以创建生成器：

- 1、通过生成器函数声明：

```
function* genFunc() { ··· }
let genObj = genFunc();
```

- 2、通过生成器函数表达式：

```
const genFunc = function* () { ··· };
let genObj = genFunc();
```

- 3、通过对象字面量中的生成器方法定义：

```
let obj = {
    * generatorMethod() {
        ...
    }
};
let genObj = obj.generatorMethod();
```

- 4、通过类定义中的生成器方法定义（可以是类声明或者类定义）：

```
class MyClass {
    * generatorMethod() {
        ...
    }
}
let myInst = new MyClass();
let genObj = myInst.generatorMethod();
```

21.2.2 生成器扮演的角色

生成器可扮演三种角色：

- 1、迭代器（数据生产者）：每一个 `yield` 都可以通过 `next()` 返回一个值，这意味着生成器可通过循环和递归生成一组值。由于生成器实现了 `Iterable` 接口（该接口在关于迭代的那一章有讲解），因此这组值可以传给 ECMAScript 6 中任何支持迭代的结构。例如： `for-of` 循环和扩展操作符（...）。

- 2、观察者（数据消费者）： `yield` 也可以从 `next()` 中得到值（通过向 `next()` 方法传入一个参数）。这意味着生成器变成了数据消费者：暂停执行直到一个新的值通过 `next()` 传入。
- 3、协同程序（数据生产者和消费者）：生成器有了暂停执行的能力，和既能成为数据生产者又能成为数据消费者的能力之后，实现协同程序（同时执行多个任务）就不需要做太多的工作了。

下一节更深入地讲解了这三种角色。

21.3 用作迭代器的生成器（数据生产）

> 在学习本节之前，你应该对 ES6 迭代熟悉，上一章讲解了更多关于迭代的知识。

正如之前讲解的，生成器对象可以是数据生产者，数据消费者或者两者都是。本节将其视为数据生产者，同时实现 `Iterable` 和 `Iterator` 接口（如下所示）。这意味着一个生成器的结果既是一个 Iterable 对象又是一个 Iterator 对象。完整的生成器对象接口将会在后面展示。

```
interface Iterable {
    [Symbol.iterator]() : Iterator;
}
interface Iterator {
    next() : IteratorResult;
    return?(value? : any) : IteratorResult;
}
interface IteratorResult {
    value : any;
    done : boolean;
}
```

生成器函数通过 `yield` 生成一组值，数据消费者通过继承自 Iterator 的方法 `next()` 来消费这些值。例如，下面的生成器函数生成值 `'a'` 和 `'b'` 。

```
function* genFunc() {
    yield 'a';
    yield 'b';
}
```

下面的交互展示了如何通过生成器对象 `genObj` 得到 yield 返回的值：

```
> let genObj = genFunc();
> genObj.next()
{ value: 'a', done: false }
> genObj.next()
{ value: 'b', done: false }
> genObj.next() // done: true => end of sequence
{ value: undefined, done: true }
```

21.3.1 迭代一个生成器的方式

由于生成器对象是可迭代的，所以 ES6 中支持迭代的语言结构都可以使用生成器对象。下面三种结构尤其重要。

首先是 `for-of` 循环：

```
for (let x of genFunc()) {
    console.log(x);
}
// Output:
// a
// b
```

其次是扩展操作符（...），将可迭代的序列转换成一个数组的元素（参考关于参数处理的那一章来获取有关这个操作符的更多信息）：

```
let arr = [...genFunc()]; // ['a', 'b']
```

第三个，解构：

```
> let [x, y] = genFunc();
> x
'a'
> y
'b'
```

21.3.2 从生成器返回

前面的生成器函数没有包含一个显示的 `return` 。隐式的 `return` 等价于返回 `undefined` 。让我们看看带有显示 `return` 的生成器：

```
function* genFuncWithReturn() {
    yield 'a';
    yield 'b';
    return 'result';
}
```

`next()` 返回的最后一个对象包含了返回值，并且对象的属性 `done` 为 `true` ：

```
> let genObjWithReturn = genFuncWithReturn();
> genObjWithReturn.next()
{ value: 'a', done: false }
> genObjWithReturn.next()
{ value: 'b', done: false }
> genObjWithReturn.next()
{ value: 'result', done: true }
```

但是，大多数做迭代的语法结构都忽略 `done` 属性里面的值：

```
for (let x of genFuncWithReturn()) {
    console.log(x);
}
// Output:
// a
// b

let arr = [...genFuncWithReturn()]; // ['a', 'b']
```

`yield*` 是一个执行生成器嵌套调用的操作符，它会考虑 `done` 属性里面的值，这在后面讲解。

21.3.3 示例：迭代属性

让我们看一个示例，该例子展示了生成器实现迭代功能是多么的方便。下面的函数， `objectEntries()` ，返回一个对象属性的迭代器：

```
function* objectEntries(obj) {
    // In ES6, you can use strings or symbols as property keys,
    // Reflect.ownKeys() retrieves both
    let propKeys = Reflect.ownKeys(obj);

    for (let propKey of propKeys) {
        yield [propKey, obj[propKey]];
    }
}
```

该函数使你能够通过 `for-of` 循环迭代对象 `jane` 的属性：

```
let jane = { first: 'Jane', last: 'Doe' };
for (let [key,value] of objectEntries(jane)) {
    console.log(`${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe
```

为了作为对比 - 一个不使用生成器的 `objectEntries()` 实现要复杂很多：

```
function objectEntries(obj) {
    let index = 0;
    let propKeys = Reflect.ownKeys(obj);

    return {
        [Symbol.iterator]() {
            return this;
        },
        next() {
            if (index < propKeys.length) {
                let key = propKeys[index];
                index++;
                return { value: [key, obj[key]] };
            } else {
                return { done: true };
            }
        }
    };
}
```

21.3.4 只能在生成器中使用 `yield`

一个值得注意的生成器的限制是只能在生成器函数中使用 yield 。也就是说，在回调函数中使用 yield 是没有效果的：

```
function* genFunc() {
    ['a', 'b'].forEach(x => yield x); // SyntaxError
}
```

不能在非生成器函数中使用 `yield` ，这就是为什么前面的代码引起了语法错误。在此种场景下，把代码改成不使用回调函数的形式是很容易的（如下所示）。但是不幸的是这不是总会奏效的。

```
function* genFunc() {
    for (let x of ['a', 'b']) {
        yield x; // OK
    }
}
```

上面的限制在后面讲解：它们使生成器更容易实现，并且兼容事件循环。

21.3.5 通过 `yield*` 嵌套（用于输出）

只能在生成器函数中使用 `yield` 。因此，如果想实现一个生成器的嵌套逻辑，需要一种从其它生成器中调用某一个生成器的方式。本节会展示这种方式，实际上这比听起来的要复杂，这就是为什么 ES6 有一个特殊的操作符 `yield*` 。现在，我只解释在两个生成器都有输出的时候 `yield*` 是如何工作的，我会在后面讲解如果包含输入的话优势如何工作的。

一个生成器是如何嵌套调用另一个生成器的？假设已经有一个生成器函数 `foo` ：

```
function* foo() {
    yield 'a';
    yield 'b';
}
```

如何在另一个生成器函数 `bar` 中调用 `foo` ？下面的方式是不顶用的！

```
function* bar() {
    yield 'x';
    foo(); // does nothing!
    yield 'y';
}
```

调用 `foo()` 返回一个对象，但是实际上并不会执行 `foo()` 。这就是为什么 ECMAScript 6 有操作符 `yield*` ，该操作符用于生成器的嵌套调用：

```
function* bar() {
    yield 'x';
    yield* foo();
    yield 'y';
}

// Collect all values yielded by bar() in an array
let arr = [...bar()];
    // ['x', 'a', 'b', 'y']
```

在内部， `yield*` 类似于：

```
function* bar() {
    yield 'x';
    for (let value of foo()) {
        yield value;
    }
    yield 'y';
}
```

`yield*` 的操作数不一定是生成器对象，也可以是可迭代的对象：

```
function* bla() {
    yield 'sequence';
    yield* ['of', 'yielded'];
    yield 'values';
}

let arr = [...bla()];
    // ['sequence', 'of', 'yielded', 'values']
```

21.3.5.1 用 `yield*` 拿到最后一个值

大多数支持迭代的语法结构都会忽略迭代的最后一个对象（ `done` 属性为 `true` ）。生成器通过 `return` 返回最后一个值。 `yield*` 表达式的值就是迭代的最后一个对象：

```
function* genFuncWithReturn() {
    yield 'a';
    yield 'b';
    return 'The result';
}
function* logReturned(genObj) {
    let result = yield* genObj;
    console.log(result); // (A)
}
```

如果想执行到行 A，首先就要迭代掉 `logReturned()` 中生成的所有值：

```
> [...logReturned(genFuncWithReturn())]
The result
[ 'a', 'b' ]
```

21.3.5.2 遍历树

用递归的方式遍历一棵树是很简单的，用传统的方式给一颗树添加一个迭代器是很复杂的。这就是为什么生成器在此处大放异彩：可以通过递归实现一个迭代器。作为一个例子，考虑下面的二叉树的数据结构。它是可迭代的，因为有一个键为 `Symbol.iterator` 的方法，该方法是一个迭代器方法，在调用的时候返回一个迭代器。

```
class BinaryTree {
    constructor(value, left=null, right=null) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    /** Prefix iteration */
    * [Symbol.iterator]() {
        yield this.value;
        if (this.left) {
            yield* this.left;
        }
        if (this.right) {
            yield* this.right;
        }
    }
}
```

下面的代码创建了一颗二叉树，并通过 `for-of` 循环遍历：

```
let tree = new BinaryTree('a',
    new BinaryTree('b',
        new BinaryTree('c'),
        new BinaryTree('d')),
    new BinaryTree('e'));

for (let x of tree) {
    console.log(x);
}
// Output:
// a
// b
// c
// d
// e
```

21.4 用作观察者的生成器（数据消费）

作为数据消费者，生成器对象遵循生成器接口的后半部分， `Observer` ：

```
interface Observer {
    next(value? : any) : void;
    return(value? : any) : void;
    throw(error) : void;
}
```

作为一个观察者，生成器处于暂停状态，直到有输入进来。有三种类型的输入，通过接口中声明的方法来传入数据：

- `next()` 传入正常的输入数据。
- `return()` 终止生成器。
- `throw()` 发出一个出错信号。

21.4.1 通过 `next()` 发送值

如果将生成器用作观察者，那么可以通过 `next()` 传入值，然后通过 `yield` 获取传入的值。

```
function* dataConsumer() {
    console.log('Started');
    console.log(`1. ${yield}`); // (A)
    console.log(`2. ${yield}`);
    return 'result';
}
```

首先，创建一个生成器对象：

```
> let genObj = dataConsumer();
```

现在调用 `genObj.next()` ，启动生成器，执行到第一个 yield 处，然后暂停。 `next()` 的返回值就是在行 A yield 后面的值（是 `undefined` ，因为 `yield` 没有操作数）。在本节，我们对 `next()` 的返回值不感兴趣，因为仅使用 `next()` 传送值，而不是获取值。

```
> genObj.next()
Started
{ value: undefined, done: false }
```

为了将值 `a` 传给第一个 yield ，值 `b` 传给第二个 yield ，再调用两次 `next()` ：

```
> genObj.next('a')
1. a
{ value: undefined, done: false }

> genObj.next('b')
2. b
{ value: 'result', done: true }
```

最后一个 `next()` 的返回值就是 `dataConsumer()` 中 return 返回的值。 `done` 为 `true` 表明生成器执行完成了。

很不幸， `next()` 是非对称的：它总是传值给当前暂停的 yield ，返回下一个 yield 的操作数。

21.4.1.1 第一个 `next()`

在把生成器当成观察者的时候，一定要注意第一次 `next()` 调用是为了启动观察者，然后才接收输入，因为第一次调用使执行流程推进到了第一个 `yield` 处。因此，不能通过第一个 `next()` 传送值 - 如果这样做了，甚至会得到一个错误：

```
> function* g() { yield }
> g().next('hello')
TypeError: attempt to send 'hello' to newborn generator
```

下面的工具函数修复了这个问题：

```
/**
 * Returns a function that, when called,
 * returns a generator object that is immediately
 * ready for input via `next()`
 */
function coroutine(generatorFunction) {
    return function (...args) {
        let generatorObject = generatorFunction(...args);
        generatorObject.next();
        return generatorObject;
    };
}
```

为了理解 `coroutine()` 是如何工作的，让我们比较一下包装了的生成器和普通的生成器：

```
const wrapped = coroutine(function* () {
    console.log(`First input: ${yield}`);
    return 'DONE';
});
const normal = function* () {
    console.log(`First input: ${yield}`);
    return 'DONE';
};
```

包装了的生成器立刻就可以接收输入了：

```
> wrapped().next('hello!')
First input: hello!
```

普通的生成器需要额外调用一次 `next()` 才能接收输入：

```
> let genObj = normal();
> genObj.next()
{ value: undefined, done: false }
> genObj.next('hello!')
First input: hello!
{ value: 'DONE', done: true }
```

21.4.2 `yield` 的结合性很低

`yield` 的结合性很低，因此没必要将它的操作数放在括号里面：

```
yield a + b + c;
```

这被看作：

```
yield (a + b + c);
```

而不是：

```
(yield a) + b + c;
```

很多操作符的结合性都比 `yield` 高，如果想把整个 `yield` 表达式用作操作数，就必须将其放在括号里面。例如，如果用不带括号的 `yield` 表达式作为加法运算的操作数，就会抛出语法错误：

```
console.log('Hello' + yield); // SyntaxError
console.log('Hello' + yield 123); // SyntaxError

console.log('Hello' + (yield)); // OK
console.log('Hello' + (yield 123)); // OK
```

如果 `yield` 表达式直接就是函数或者方法的参数，可以不用括号：

```
foo(yield 'a', yield 'b');
```

如果用作赋值的右操作数，也可以不用括号：

```
let input = yield;
```

21.4.2.1 ES6 语法中的 `yield`

从下面的来自于 ECMAScript 6 规范的语法规则中可以看出用括号包裹 `yield` 是否有必要。这些规则描述了如何解析表达式。我按照一般性（底结合性，底优先级）到特殊性（高结合性，高优先级）的顺序列出这些规则。在任何需要使用某种表达式的地方，也可以使用该表达式更特殊的版本。反过来就不行了。下面的层级结构以 `ParenthesizedExpression` 结束，这意味着如果将 `yield` 表达式放在括号里面，就可以在任何表达式的任何位置正常使用了。

```
Expression :
    AssignmentExpression
    Expression , AssignmentExpression
AssignmentExpression :
    ConditionalExpression
    YieldExpression
    ArrowFunction
    LeftHandSideExpression = AssignmentExpression
    LeftHandSideExpression AssignmentOperator AssignmentExpressi
on

...

AdditiveExpression :
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression
MultiplicativeExpression :
    UnaryExpression
    MultiplicativeExpression MultiplicativeOperator UnaryExpress
ion

...

PrimaryExpression :
    this
    IdentifierReference
    Literal
    ArrayLiteral
    ObjectLiteral
    FunctionExpression
    ClassExpression
    GeneratorExpression
    RegularExpressionLiteral
    TemplateLiteral
    ParenthesizedExpression
ParenthesizedExpression :
    ( Expression )
```

`AdditiveExpression` 的操作数是 `AdditiveExpression` 和 `MultiplicativeExpression` 。因此，使用（更特殊的） `ParenthesizedExpression` 作为操作数是可以的，但是使用（更一般的） `YieldExpression` 就不行了。

21.4.3 `return()` 和 `throw()`

让我们回顾一下 `next(x)` 是如何工作的（在第一次调用之后）：

- 1、生成器此时暂停在一个 `yield` 操作符处。

- 2、传递值 `x` 给上面的 `yield` ，这意味着 `yield` 表达式的值就为 `x` 。
- 3、执行到下一个 `yield` 或者 `return` ：
    - `yield x` 使 `next()` 返回 `{ value: x, done: false }`
    - `return x` 使 `next()` 返回 `{ value: x, done: true }`

`return()` 和 `throw()` 类似于 `next()` ，但是在第二步中做了不一样的事情：

- `return(x)` 在 `yield` 处执行 `return x` 。
- `throw(x)` 在 `yield` 处执行 `throw x` 。

21.4.4 `return()` 终止生成器

`return()` 在 `yield` 处执行 `return` ，使该处 `yield` 成为生成器的最后一次暂停点。让我们使用下面的生成器函数来看看这是怎么工作的。

```
function* genFunc1() {
    try {
        console.log('Started');
        yield; // (A)
    } finally {
        console.log('Exiting');
    }
}
```

在下面的交互执行过程中，首先使用 `next()` 启动生成器，执行到行 A 的 yield 处暂停。然后在那个位置通过 `return()` 返回。

```
> let genObj1 = genFunc1();
> genObj1.next()
Started
{ value: undefined, done: false }
> genObj1.return('Result')
Exiting
{ value: 'Result', done: true }
```

21.4.4.1 组织终止

如果在 `finally` 子句中使用 yield ，则可以阻止 `return()` 终止生成器（也可以在 `finally` 子句中使用 `return` 语句来阻止）。

```
function* genFunc2() {
    try {
        console.log('Started');
        yield;
    } finally {
        yield 'Not done, yet!';
    }
}
```

这次， `return()` 并不会退出生成器函数。相应地，它返回的对象的 `done` 属性为 `false` 。

```
> let genObj2 = genFunc2();

> genObj2.next()
Started
{ value: undefined, done: false }

> genObj2.return('Result')
{ value: 'Not done, yet!', done: false }
```

可以再调用一次 `next()` 。类似于非生成器函数，生成器函数的返回值是在进入 `finally` 子句之前放入队列中的值。

```
> genObj2.next()
{ value: 'Result', done: true }
```

21.4.4.2 从新生的生成器返回

从新生（尚未启动）的生成器中返回值是可以的：

```
> function* genFunc() {}
> genFunc().return('yes')
{ value: 'yes', done: true }
```

> 深入阅读
>
> 关于迭代的那一章中有一节详细讲解了关闭迭代器和生成器。

21.4.5 `throw()` 抛出错误

`throw()` 在 `yield` 处抛出异常，使该 `yield` 成为生成器最后的暂停点。让我们通过下面的生成器函数看看这是如何工作的。

```
function* genFunc1() {
    try {
        console.log('Started');
        yield; // (A)
    } catch (error) {
        console.log('Caught: ' + error);
    }
}
```

在下面的交互过程中，首先调用 `next()` 启动生成器，执行到行 A 的 `yield` 处暂停，然后在那个位置抛出异常。

```
> let genObj1 = genFunc1();

> genObj1.next()
Started
{ value: undefined, done: false }

> genObj1.throw(new Error('Problem!'))
Caught: Error: Problem!
{ value: undefined, done: true }
```

`throw()`（最后一行展示）的返回值来自于离开生成器函数时隐式的 `return`。

21.4.5.1 未捕获的异常

如果没有在生成器内部捕获异常，就会通过 `throw()` 抛出来。例如，下面的生成器函数不捕获异常：

```
function* genFunc2() {
    console.log('Started');
    yield; // (A)
}
```

如果在行 A 处用 `throw()` 抛出一个 `Error` 的实例，那么该生成器函数自身就会抛出这个错误：

```
> let genObj2 = genFunc2();
> genObj2.next()
Started
{ value: undefined, done: false }
> genObj2.throw(new Error('Problem!'))
Error: Problem!
```

21.4.5.2 从新生的生成器抛出异常

在一个新生（尚未启动）的生成器中抛出异常是可以的：

```
> function* genFunc() {}
> genFunc().throw(new Error('Problem!'))
Error: Problem!
```

21.4.6 示例：处理异步推送的数据

作为观察者的生成器在等待输入的时候暂停的特性使其能非常完美地按需处理异步数据。可以创建一个生成器链来处理这些异步的数据，下面描述了生成器链的要素：

- 生成器链中的每一个成员（除了最后一个）都有一个 `target` 参数。这些成员通过 `yield` 获取数据，通过 `target.next()` 发送数据。
- 生成器链的最后一个成员没有 `target` 参数，仅获取数据。

整个生成器前面有一个非生成器函数，该函数发出异步请求，并将请求结果通过 `next()` 推送给生成器链。

作为一个例子，让我们将一组生成器组成一条链来处理异步读取的文件数据。

> 本例中的代码位于文件 generator-examples/node/readlines.js 中。必须通过 `babel-node` 执行。

下面的代码创建这条链：它包含生成器 `splitLines` ， `numberLines` 和 `printLines` 。数据通过非生成器函数 `readFile()` 推送给生成器链。

```
readFile(fileName, splitLines(numberLines(printLines())));
```

在我展示这些函数的代码的时候，会讲解它们都做了些什么。

正如之前讲解的，如果生成器通过 `yield` 接收输入数据，生成器对象上的第一次 `next()` 调用不会做任何事。这就是为什么我要使用之前展示的辅助方法 `coroutine()` 来创建协同程序。该辅助方法帮助我们执行第一次的 `next()` 调用。

`readFile()` 是一个非生成器函数，启动了整个流程：

```
import {createReadStream} from 'fs';

/**
 * Create an asynchronous ReadStream for the file whose name
 * is `fileName` and feed it to the generator object `target`.
 *
 * @see ReadStream https://nodejs.org/api/fs.html#fs_class_fs_re
adstream
 */
function readFile(fileName, target) {
    let readStream = createReadStream(fileName,
        { encoding: 'utf8', bufferSize: 1024 });
    readStream.on('data', buffer => {
        let str = buffer.toString('utf8');
        target.next(str);
    });
    readStream.on('end', () => {
        // Signal end of output sequence
        target.return();
    });
}
```

生成器链以 `splitLines` 开始：

```
/**
 * Turns a sequence of text chunks into a sequence of lines
 * (where lines are separated by newlines)
 */
const splitLines = coroutine(function* (target) {
    let previous = '';
    try {
        while (true) {
            previous += yield;
            let eolIndex;
            while ((eolIndex = previous.indexOf('\n')) >= 0) {
                let line = previous.slice(0, eolIndex);
                target.next(line);
                previous = previous.slice(eolIndex+1);
            }
        }
    } finally {
        // Handle the end of the input sequence
        // (signaled via `return()`)
        if (previous.length > 0) {
            target.next(previous);
        }
        // Signal end of output sequence
        target.return();
    }
});
```

请注意：

- `readFile()` 使用生成器对象方法 `return()` 来通知生成器链它发送的数据块结束了。
- 在 `splitLines` 的无限循环里通过 `yield` 等待输入数据的时候，`readFile()` 发出数据结束信号。`return()` 使其跳出那个无限循环。
- `splitLines` 使用 `finally` 子句处理扫尾工作。

下一个生成器是 `numberLines` ：

```
/**
 * Prefixes numbers to a sequence of lines
 */
const numberLines = coroutine(function* (target) {
    try {
        for (let lineNo = 0; ; lineNo++) {
            let line = yield;
            target.next(`${lineNo}: ${line}`);
        }
    } finally {
        // Signal end of output sequence
        target.return();
    }
});
```

最后一个生成器是 `printLines` ：

```
/**
 * Receives a sequence of lines (without newlines)
 * and logs them (adding newlines).
 */
const printLines = coroutine(function* () {
    while (true) {
        let line = yield;
        console.log(line);
    }
});
```

上述代码很奇妙，所有的事情都是惰性的（按需的）：在数据到达的时候，拆分行，加上行号，打印出来；在能够打印之前并不需要等到所有文本都读取完毕。

21.4.7 yield* ：所有细节

`yield*` 可以实现从一个生成器函数中（调用者）调用另一个生成器函数（被调用者）的功能。

到目前为止，我们仅看到了 `yield` 的一个方面：将 `yield` 中的值从被调用者传给调用者。既然我们对生成器接收输入感兴趣，那么另一方面就变得有意义了：`yield*` 也将调用者接收到的输入数据转发给被调用者。

首先我将展示如何在 JavaScript 中实现 `yield*`，以此来讲解 `yield*` 的完整语义。然后给出简单的例子，调用者通过 `next()` 、 `return()` 和 `throw()` 接收到输入数据，转发给被调用者。

下面的语句：

```
let yieldStarResult = yield* calleeFunc();
```

大致等价于：

```
let yieldStarResult;

let calleeObj = calleeFunc();
let prevReceived = undefined;
while (true) {
    try {
        // Forward input previously received
        let {value,done} = calleeObj.next(prevReceived);
        if (done) {
            yieldStarResult = value;
            break;
        }
        prevReceived = yield value;
    } catch (e) {
        // Pretend `return` can be caught like an exception
        if (e instanceof Return) {
            // Forward input received via return()
            calleeObj.return(e.returnedValue);
            return e.returnedValue; // “re-throw”
        } else {
            // Forward input received via throw()
            calleeObj.throw(e); // may throw
        }
    }
}
```

为了保持简单，上面的代码没有考虑几件事：

- `yield*` 的操作数可以是任何可迭代的值。
- `return()` 和 `throw()` 是可选的迭代器方法。应该仅在它们存在的时候才调用。
- 如果接收到异常， `throw()` 不存在，但是有 `return()` ，此时调用 `return()` （在抛出异常之前），让 `calleeObject` 有机会做清理工作。
- `calleeObj` 可以通过返回一个 `done` 属性为 `false` 的对象来拒绝关闭。

然后调用者也不得不拒绝关闭，`yield*` 继续迭代。

21.4.7.1 示例：用 `yield*` 转发 `next()` 传入的值

下面的生成器函数 `caller()` 通过 `yield*` 调用生成器函数 `callee()` 。

```
function* callee() {
    console.log('callee: ' + (yield));
}
function* caller() {
    while (true) {
        yield* callee();
    }
}
```

`callee` 打印 `next()` 传入的值，这样就可以检查传给 `caller` 的值是 `a` 还是 `b` 。

```
> let callerObj = caller();

> callerObj.next() // start
{ value: undefined, done: false }

> callerObj.next('a')
callee: a
{ value: undefined, done: false }

> callerObj.next('b')
callee: b
{ value: undefined, done: false }
```

21.4.7.2 示例：用 `yield*` 转发 `throw()` 抛出的异常

用下面一段代码来展示当 `yield*` 代理另一个生成器的时候，`throw()` 是如何工作的。

```
function* callee() {
    try {
        yield 'b'; // (A)
        yield 'c';
    } finally {
        console.log('finally callee');
    }
}
function* caller() {
    try {
        yield 'a';
        yield* callee();
        yield 'd';
    } catch (e) {
        console.log('[caller] ' + e);
    }
}
```

首先创建一个生成器对象，然后执行到行 A 。

```
> let genObj = caller();

> genObj.next().value
'a'
> genObj.next().value
'b'
```

在行 A ，抛出一个异常：

```
> genObj.throw(new Error('Problem!'))
finally callee
[caller] Error: Problem!
{ value: undefined, done: true }
```

`callee` 并不会捕获该异常，异常会传进 `caller` ，所以会在 `caller` 执行完之前打印出异常。

21.4.7.3 用 `yield*` 转发 `return()` 返回值

用下面一段代码来展示当 `yield*` 代理另一个生成器的时候， `return()` 是如何工作的。

```
function* callee() {
    try {
        yield 'b';
        yield 'c';
    } finally {
        console.log('finally callee');
    }
}
function* caller() {
    try {
        yield 'a';
        yield* callee();
        yield 'd';
    } finally {
        console.log('finally caller');
    }
}
```

解构在拿到了需要的元素之后，如果迭代器还未迭代完，那么就会通过 `return()` 关闭这个迭代器：

```
let [x, y] = caller(); // ['a', 'b']

// Output:
// finally callee
// finally caller
```

有趣的是， `return()` 被送入 `caller` ，然后转发给 `callee` （ `callee` 更早结束 ），但是之后也会终止 `caller` （这是调用 `return()` 的人所希望看到的）。也就是说， `return` 的传播和异常很类似。

> 理解 `return()` 的小提示
>
> 为了理解 `return()` ，这样问自己是很有帮助的：如果我把 callee 函数中代码复制粘贴到 caller 函数中，会发生什么。

21.5 生成器用作协同程序（多任务协作）

我们已经学习了生成器用作数据源和数据接收器。对于很多应用来说，最好严格区分这两个角色，因为这会使事情变得更加简单。本节描述了整个生成器接口（混合了两种角色），并在一个场景中两种角色都需要：多任务协作，任务既要能够发送信息，又要能够接收信息。

21.5.1 完整的生成器接口

完整的生成器对象接口，`Generator`，既处理输出又处理输入：

```
interface Generator {
    next(value? : any) : IteratorResult;
    throw(value? : any) : IteratorResult;
    return(value? : any) : IteratorResult;
}
interface IteratorResult {
    value : any;
    done : boolean;
}
```

> 该接口在规范文档的“Properties of Generator Prototype”节描述了。

`Generator` 接口混合了两种之前章节见过的接口：用于输出的 `Iterator` 接口和用于输入的 `Observer` 接口。

```
interface Iterator { // data producer
    next() : IteratorResult;
    return?(value? : any) : IteratorResult;
}

interface Observer { // data consumer
    next(value? : any) : void;
    return(value? : any) : void;
    throw(error) : void;
}
```

21.5.2 多任务协作

多任务协作场景需要生成器既能接收输入又能处理输出。在理解它的工作原理之前，先复习一下当前 JavaScript 中的并行方式。

JavaScript 运行在单进程中，有两种方式可以突破这个限制：

- 多重处理（Multiprocessing）：Web Worker 使你可以在多个进程中运行

JavaScript 代码。数据共享是多重处理的最大陷阱。 Web Worker 通过不共享任何数据来避免这个问题。也就是说，如果你希望某个 Web Worker 能拿到数据，就必须传入数据的副本或者将数据转移给该 Web Worker （在这之后你再也不能在当前进程中访问该数据了）。
- 多任务协作（ Cooperative multitasking ）：有大量的模式和库尝试处理多任务协作。运行多个任务的时候，在某一刻只能有一个任务执行。每一个任务必须显示地暂停自己，在任务切换的时候释放控制权。在这些尝试中，数据经常在任务之间共享。但是由于需要显示地暂停，就引入了一些风险。

有两种应用场景从多任务协作中受益，因为两种场景包含了通常按序执行的控制流，但是偶尔还是会暂停的。

- **Streams** ：任务会按序处理数据流，并且在没有数据的时候暂停。
  - 对于二进制数据， WHATWG 正致力于一个建议标准，该标准基于回调和 Promise 。
  - 对于数据流，通信顺序进程（ CSP ）是一个有趣的解决方案。基于生成器的 CSP 库在本章后续部分有讲述。
- 异步计算 ：在执行一个长时间运行的计算的时候，任务会被阻塞（暂停），直到计算完成。
  - 在 JavaScript 中， Promise 成为了处理异步计算的流行方式。 ES6 对其提供了支持。下节讲解了生成器是如何让 Promise 的使用变得简单。

21.5.2.1 通过生成器简化异步计算

几个基于 Promise 的库通过生成器简化异步代码。生成器用作 Promise 的接收器是非常完美的，因为生成器可以暂停，直到传入计算结果。

下面的例子展示了使用 T.J. Holowaychuk 的 co 库 之后写出来的代码的样子。我们需要两个库（如果通过 `babel-node` 运行 Node.js 代码）。

```
require('isomorphic-fetch'); // polyfill
let co = require('co');
```

`co` 是用于多任务协作的实际可用的库， `isomorphic-fetch` 是新的基于 `fetch` API （ XMLHttpRequest 的替代方案；阅读 Jake Archibald 的“ That's so fetch! ”获取更多相关信息）的一个 polyfill 。 `fetch` 使得写一个返回指定 `url` 的文件的文本内容的函数 `getFile` 变得容易：

```
function getFile(url) {
    return fetch(url)
        .then(request => request.text());
}
```

我们现在具备了使用 `co` 的所有条件。下面的代码读取两个文件的文本内容，解析里面的 JSON 数据，并打印结果。

```
co(function* () {
    try {
        let [croftStr, bondStr] = yield Promise.all([  // (A)
            getFile('http://localhost:8000/croft.json'),
            getFile('http://localhost:8000/bond.json'),
        ]);
        let croftJson = JSON.parse(croftStr);
        let bondJson = JSON.parse(bondStr);

        console.log(croftJson);
        console.log(bondJson);
    } catch (e) {
        console.log('Failure to read: ' + e);
    }
});
```

注意这段代码看起来是多么漂亮的同步代码，即使在行 A 执行了异步调用。作为执行异步任务的生成器，通过 yield 传给整个任务流程的调度者 co 一个 Promise 对象。 yield 使得生成器暂停下来。一旦 Promise 返回结果，调度者 co 就会通过 `next()` 唤醒生成器，并传入结果。一个简单版本的 co 看起来像下面这样。

```
function co(genFunc) {
    let genObj = genFunc();
    run();

    function run(promiseResult = undefined) {
        let {value,done} = genObj.next(promiseResult);
        if (!done) {
            // A Promise was yielded
            value
            .then(result => run(result))
            .catch(error => {
                genObj.throw(error);
            });
        }
    }
}
```

21.5.3 基于生成器的多任务协作的限制

协同程序是没有限制的多任务协作程序：在协同程序内部，任何函数都可以暂停整个协同程序（函数自身的激活，函数调用者的激活，调用者的调用者，等等）。

相对而言，你只能在生成器函数的直接内部暂停该生成器，并且只能让当前的函数变为非活跃状态。由于这些限制，生成器有时又被叫做浅协同程序（ shallow coroutines ）。

21.5.3.1 生成器限制的好处

生成器的限制有两个主要的好处：

- 生成器和事件循环兼容，后者在浏览器中提供了简单的多任务协作。我会马上讲解详细的内容。
- 生成器实现起来相当简单，因为仅需要暂停一个活跃的函数，并且浏览器可以继续使用事件循环。

JavaScript 已经有一种非常简单的多任务协作方式：事件循环，安排任务队列中任务的执行。每个任务都开始于某个函数的调用，结束于该函数调用的完成。事件，`setTimeout()` 和其它一些机制添加任务到队列中。

> 这是事件循环的简单解释，对于目前而言足够了。如果你对细节感兴趣，参考关于异步编程的那一章。

这种方式的多任务确保了一件重要的事情：运行的完整性；每一个函数在执行完之前都不会被打断。函数成为了事务，可以执行完整的逻辑，不用暴露出来它们操作的处于某个中间状态的数据。并发访问共享数据使得多任务变得复杂起来，而在 JavaScript 的并发模型中这是不允许的。这就是为什么运行的完整性是一个优点。

然而，协同程序妨碍了运行的完整性，因为任何函数都可以暂停它的调用者。例如，下面的逻辑由多步组成：

```
step1(sharedData);
step2(sharedData);
lastStep(sharedData);
```

如果 `step2` 暂停了逻辑，其它任务就可以先于当前任务的最后一步执行。这些任务可能包含当前应用的其它部分，这些部分可以访问到处于未完成状态的 `sharedData` 。生成器保护了执行的完整性，它只暂停自身，然后返回调用者。

`co` 和类似的库提供了协同程序的强大能力，而摒弃掉了协同程序的缺陷：

- 给任务提供了调度器，调度器通过生成器定义。
- 任务“就是”生成器，因此可以被完全暂停。
- 如果通过 `yield*` 嵌套调用（生成器）函数，那么该函数只具备暂停能力。调用者可以控制函数的暂停。

21.6 生成器实例

本节给出几个例子，用于说明生成器可以用来干什么。

> 下面的 GitHub 仓库包含了示例代码： generator-examples

21.6.1 通过生成器实现迭代器

在关于迭代器的章节，我"顺手"实现了几个迭代器。在本节，我用生成器来实现它们。

21.6.1.1 可迭代的选择器 `take()`

`take()` 将一组可迭代的值（可能是无限的）转换成一个长度 `n` 的序列：

```
function* take(n, iterable) {
    for (let x of iterable) {
        if (n <= 0) return;
        n--;
        yield x;
    }
}
```

下面是使用它的一个例子：

```
let arr = ['a', 'b', 'c', 'd'];
for (let x of take(2, arr)) {
    console.log(x);
}
// Output:
// a
// b
```

一个不使用生成器实现的 `take()` 版本会更加复杂：

```
function take(n, iterable) {
    let iter = iterable[Symbol.iterator]();
    return {
        [Symbol.iterator]() {
            return this;
        },
        next() {
            if (n > 0) {
                n--;
                return iter.next();
            } else {
                maybeCloseIterator(iter);
                return { done: true };
            }
        },
        return() {
            n = 0;
            maybeCloseIterator(iter);
        }
    };
}
function maybeCloseIterator(iterator) {
    if (typeof iterator.return === 'function') {
        iterator.return();
    }
}
```

注意迭代选择器 zip() 通过生成器实现并不会获益多少，因为要使用多个迭代器，不能使用 `for-of` 循环。

21.6.1.2 无限迭代器

`naturalNumbers()` 返回一个迭代所有自然数的迭代器：

```
function* naturalNumbers() {
    for (let n=0;; n++) {
        yield n;
    }
}
```

这种函数通常和选择器结合使用：

```
for (let x of take(3, naturalNumbers())) {
    console.log(x);
}
// Output
// 0
// 1
// 2
```

下面是非生成器版本的实现，你可以对比一下：

```
function naturalNumbers() {
    let n = 0;
    return {
        [Symbol.iterator]() {
            return this;
        },
        next() {
            return { value: n++ };
        }
    }
}
```

21.6.1.3 灵感来自于数组的迭代选择器： `map` ， `filter`

数组可以通过 `map` 和 `filter` 方法转换。这些方法可以一般化为把可迭代对象作为输入，然后输出另一个可迭代对象。

21.6.1.3.1 一般化的 `map()`

下面是 `map()` 的一般化版本：

```
function* map(iterable, mapFunc) {
    for (let x of iterable) {
        yield mapFunc(x);
    }
}
```

处理无限迭代对象的 `map()` ：

```
> [...take(4, map(naturalNumbers(), x => x * x))]
[ 0, 1, 4, 9 ]
```

21.6.1.3.2 一般化的 `filter()`

下面是 `filter()` 的一般化版本：

```
function* filter(iterable, filterFunc) {
    for (let x of iterable) {
        if (filterFunc(x)) {
            yield x;
        }
    }
}
```

处理无限迭代对象的 `filter()` ：

```
> [...take(4, filter(naturalNumbers(), x => (x % 2) === 0))]
[ 0, 2, 4, 6 ]
```

21.6.2 用于延迟执行的生成器

接下来的两个例子展示生成器如何用于处理字符流。

- 输入是字符流。
- 第一步 - 分词（字符→单词）：字符组合成单词，就是满足正则表达式 `/^[A-Za-z0-9]$/` 的字符串。忽略非单词的字符，但是这些字符分割单词。这一步的输入是字符流，输出是单词流。
- 第二步 - 提取数字（单词→数字）：仅保留匹配正则 `/^[0-9]+$/` 的单词，并将它们转换成数字。
- 第三步 - 添加数字（数字→数字）：对于每个接收到的数字，返回到目前为止接收到的所有数字之和。

所有内容都延迟执行（增量地和按需地），很平滑优雅：接收到第一个字符之后就立即开始计算。例如，我们不需要等到接受完所有字符才去拿第一个单词。

21.6.2.1 延迟拉取（用作迭代器的生成器）

用生成器实现的延迟拉取以如下的方式工作。实现步骤一到三的三个生成器像下面这样链式调用：

```
addNumbers(extractNumbers(tokenize(CHARS)))
```

链中的每个成员从数据源中拉取数据，然后生产出另一组数据。整个过程开始于 `tokenize` ，它的数据源就是字符串 CHARS 。

21.6.2.1.1 第一步 - 分词

下面的技巧使得代码更简单一点儿：迭代器的最后一个结果（属性 `done` 是 `false` ）设置为结束标记 END_OF_SEQUENCE 。

```
/**
 * Returns an iterable that transforms the input sequence
 * of characters into an output sequence of words.
 */
function* tokenize(chars) {
    let iterator = chars[Symbol.iterator]();
    let ch;
    do {
        ch = getNextItem(iterator); // (A)
        if (isWordChar(ch)) {
            let word = '';
            do {
                word += ch;
                ch = getNextItem(iterator); // (B)
            } while (isWordChar(ch));
            yield word; // (C)
        }
        // Ignore all other characters
    } while (ch !== END_OF_SEQUENCE);
}
const END_OF_SEQUENCE = Symbol();
function getNextItem(iterator) {
    let {value,done} = iterator.next();
    return done ? END_OF_SEQUENCE : value;
}
function isWordChar(ch) {
    return typeof ch === 'string' && /^[A-Za-z0-9]$/.test(ch);
}
```

该生成器是如何延迟的？当通过 `next()` 请求一个值的时候，该生成器就从 `iterator` （行 A 和行 B ）中拉取数据，处理之后就放在 yield 后面返回出去（行 C ）。然后生成器函数暂停，直到下一次数据请求的到来。这意味着这种分词在第一个字符就绪的时候就开始了，这对流（ stream ）很方便。

让我们尝试一下分词。注意空格和点符号不是单词，忽略它们，但是它们分割单词。我们利用一个事实：字符串就是字符（ unicode 码点）的迭代。`tokenize()` 的结果就是一组单词的迭代，可以通过扩展操作符（...）将其转换为数组。

```
> [...tokenize('2 apples and 5 oranges.')]
[ '2', 'apples', 'and', '5', 'oranges' ]
```

21.6.2.1.2 第二步 - 提取数字

这一步相当简单，仅将包含数字的单词（通过 `Number()` 转换成数字类型）放在 `yield` 之后。

```
/**
 * Returns an iterable that filters the input sequence
 * of words and only yields those that are numbers.
 */
function* extractNumbers(words) {
    for (let word of words) {
        if (/^[0-9]+$/.test(word)) {
            yield Number(word);
        }
    }
}
```

可以再次看到延迟：如果通过 `next()` 请求一个数字，那么在 `words` 中遇到一个数字单词的时候，就会立马获取到（通过 `yield` ）这个数字。

让我们从一组单词中获取数字：

```
> [...extractNumbers(['hello', '123', 'world', '45'])]
[ 123, 45 ]
```

注意字符串转换成了数字。

21.6.2.1.3 第三步 - 数字相加

```
/**
 * Returns an iterable that contains, for each number in
 * `numbers`, the total sum of numbers encountered so far.
 * For example: 7, 4, -1 --> 7, 11, 10
 */
function* addNumbers(numbers) {
    let result = 0;
    for (let n of numbers) {
        result += n;
        yield result;
    }
}
```

尝试一个简单的例子：

```
> [...addNumbers([5, -2, 12])]
[ 5, 3, 15 ]
```

21.6.2.1.4 获取输出

生成器链本身并不会产生输出。我们需要通过扩展操作符主动地获得输出数据：

```
const CHARS = '2 apples and 5 oranges.';
const CHAIN = addNumbers(extractNumbers(tokenize(CHARS)));
console.log([...CHAIN]);
    // [ 2, 7 ]
```

辅助函数 `logAndYield` 让我们能够判断出是否真的延迟执行了：

```
function* logAndYield(iterable, prefix='') {
    for (let item of iterable) {
        console.log(prefix + item);
        yield item;
    }
}

const CHAIN2 = logAndYield(addNumbers(extractNumbers(tokenize(lo
gAndYield(CHARS)\
))), '-> ');
[...CHAIN2];

// Output:
// 2
//
// -> 2
// a
// p
// p
// l
// e
// s
//
// a
// n
// d
//
// 5
//
// -> 7
// o
// r
// a
// n
// g
// e
// s
// .
```

上面的输出表明 `addNumbers` 在接收到 `'2'` 和 `' '` 的时候就生成了一个结果。

21.6.2.2 延迟推入数据（用作观察者的生成器）

将前面基于输出的逻辑改成一个基于输入的例子不需要做太多事情。步骤是一样的。但是不是以获取数据而结束，而是以输入数据而开始。

正如之前讲解的，如果生成器通过 `yield` 接收到输入，那么该生成器上的第一次 `next()` 调用什么也不干。这就是为什么我要使用之前展示的辅助函数 `coroutine()` 来在此处创建协同程序，它帮助我们首次调用 `next()` 。

下面的函数 `send()` 完成数据推入。

```
/**
 * Pushes the items of `iterable` into `sink`, a generator.
 * It uses the generator method `next()` to do so.
 */
function send(iterable, sink) {
    for (let x of iterable) {
        sink.next(x);
    }
    sink.return(); // signal end of stream
}
```

当生成器处理流的时候，它需要知道流的结束点，以便于正确地做清理工作。对于拉取数据，我们通过一个特殊的流终结符来做这件事情。对于推入数据，流的结束信号通过 `return()` 发出。

让我们通过一个生成器来测试 `send()` ，该生成器只是简单地打印出接收到的数据：

```
/**
 * This generator logs everything that it receives via `next()`.
 */
const logItems = coroutine(function* () {
    try {
        while (true) {
            let item = yield; // receive item via `next()`
            console.log(item);
        }
    } finally {
        console.log('DONE');
    }
});
```

通过一个字符串（可以迭代其中的 unicode 码点）给 `logItems()` 传入三个字符。

```
> send('abc', logItems());
a
b
c
DONE
```

21.6.2.2.1 第一步 **-** 分词

注意生成器是如何在两个 `finally` 子句中处理流结束的（通过 `return()` 发出信号）。我们依赖于 `return()` 信号被送给两个 `yield` 中的一个。否则，该生成器将永不会结束，因为从行 A 开始的无限循环不会停止。

```
/**
 * Receives a sequence of characters (via the generator object
 * method `next()`), groups them into words and pushes them
 * into the generator `sink`.
 */
const tokenize = coroutine(function* (sink) {
    try {
        while (true) { // (A)
            let ch = yield; // (B)
            if (isWordChar(ch)) {
                // A word has started
                let word = '';
                try {
                    do {
                        word += ch;
                        ch = yield; // (C)
                    } while (isWordChar(ch));
                } finally {
                    // The word is finished.
                    // We get here if
                    // - the loop terminates normally
                    // - the loop is terminated via `return()` i
n line C
                    sink.next(word); // (D)
                }
            }
            // Ignore all other characters
        }
    } finally {
        // We only get here if the infinite loop is terminated
        // via `return()` (in line B or C).
        // Forward `return()` to `sink` so that it is also
        // aware of the end of stream.
        sink.return();
    }
});

function isWordChar(ch) {
    return /^[A-Za-z0-9]$/.test(ch);
}
```

这一次，该延迟通过数据推入来驱动：当生成器接收到能够构成一个单词的字符数时（在行 C ），就会把该单词推入 `sink` （行 D ）。也就是说，生成器不会等到接收到所有的字符才开始工作。

`tokenize()` 显示了生成器作为线性状态机的实现，工作得很好。在此场景下，状态机有两种状态：“在单词内部”和“没在单词内部”。

让我们对一个字符串分词：

```
> send('2 apples and 5 oranges.', tokenize(logItems()));
2
apples
and
5
oranges
```

21.6.2.2.2 第二步 - 获取数字

这一步很直接。

```
/**
 * Receives a sequence of strings (via the generator object
 * method `next()`) and pushes only those strings to the generat
or
 * `sink` that are “numbers” (consist only of decimal digits).
 */
const extractNumbers = coroutine(function* (sink) {
    try {
        while (true) {
            let word = yield;
            if (/^[0-9]+$/.test(word)) {
                sink.next(Number(word));
            }
        }
    } finally {
        // Only reached via `return()`, forward.
        sink.return();
    }
});
```

同样是延迟的：当遇到数字的时候，就推入 `sink` 。

让我们从一组单词中获取数字：

```
> send(['hello', '123', 'world', '45'], extractNumbers(logItems(
)));
123
45
DONE
```

注意输入是一个字符串序列，输出是一个数字序列。

21.6.2.2.3 第三步 - 数字求和

这一次，通过推入单个值来响应流的结束，然后关闭 sink 。

```
/**
 * Receives a sequence of numbers (via the generator object
 * method `next()`). For each number, it pushes the total sum
 * so far to the generator `sink`.
 */
const addNumbers = coroutine(function* (sink) {
    let sum = 0;
    try {
        while (true) {
            sum += yield;
            sink.next(sum);
        }
    } finally {
        // We received an end-of-stream
        sink.return(); // signal end of stream
    }
});
```

让我们试一下这个生成器：

```
> send([5, -2, 12], addNumbers(logItems()));
5
3
15
DONE
```

21.6.2.2.4 推入输入数据

生成器链以 `tokenize` 开始，以 `logItems` 结束，并且打印出所有接收到的东西。我们通过 `send` 推入一系列字符到生成器链：

```
const INPUT = '2 apples and 5 oranges.';
const CHAIN = tokenize(extractNumbers(addNumbers(logItems())));
send(INPUT, CHAIN);

// Output
// 2
// 7
// DONE
```

下面的代码证明整个处理过程真的是延迟发生的：

```
const CHAIN2 = tokenize(extractNumbers(addNumbers(logItems({ pre
fix: '-> ' }))));
send(INPUT, CHAIN2, { log: true });

// Output
// 2
//
// -> 2
// a
// p
// p
// l
// e
// s
//
// a
// n
// d
//
// 5
//
// -> 7
// o
// r
// a
// n
// g
// e
// s
// .
// DONE
```

打印出来的内容显示 `addNumbers` 在接收到 `'2'` 和 `''` 的时候就立马产生一个结果。

21.6.3 通过生成器实现的多任务协作

21.6.3.1 推入长时间运行的任务

在本例中，我们创建一个在 web 页面显示的计数器。我们不断地优化最初的版本，直到得到一个多任务协作的版本，并且不会阻塞主线程和用户界面。

下面是 web 页面的一部分，在其中展示计数器：

```
<body>
    Counter: <span id="counter"></span>
</body>
```

下面的函数不停地计数：

```
function countUp(start = 0) {
    const counterSpan = document.querySelector('#counter');
    while (true) {
        counterSpan.textContent = String(start);
        start++;
    }
}
```

如果你运行这个函数，它会完全阻塞它所运行在的用户界面线程，并且失去响应。

让我们用生成器实现同样的功能，该生成器通过 `yield` 定期暂停（后面会展示一个用于运行该生成器的调度函数）：

```
function* countUp(start = 0) {
    const counterSpan = document.querySelector('#counter');
    while (true) {
        counterSpan.textContent = String(start);
        start++;
        yield; // pause
    }
}
```

让我们稍微地改进一下。把用户界面的更新挪到另一个生成器 `displayCounter` 里面，通过 `yield*` 调用该生成器。由于是一个生成器函数，所以也能够暂停。

```
function* countUp(start = 0) {
    while (true) {
        start++;
        yield* displayCounter(start);
    }
}
function* displayCounter(counter) {
    const counterSpan = document.querySelector('#counter');
    counterSpan.textContent = String(counter);
    yield; // pause
}
```

最后，下面是一个调度函数，可以用它来调用 `countUp()` 。生成器执行的每一步都由一个独立的任务处理，该任务通过 `setTimeout()` 创建。这意味着用户界面可以调度其它任务，并且保持可响应。

```
function run(generatorObject) {
    if (!generatorObject.next().done) {
        // Add a new task to the event queue
        setTimeout(function () {
            run(generatorObject);
        }, 1000);
    }
}
```

有了 `run` 的帮助，我们得到了一个近乎无尽的计数器，并且不会阻塞用户界面：

```
run(countUp());
```

你可以在线运行这个例子。

21.6.3.2 生成器版本的多任务协作和 Node.js 风格的回调

如果你调用一个生成器函数（或者方法），该函数（或方法）并不能访问到它的生成器对象；它的 `this` 与它是普通函数时的 `this` 一样。一种变通方法就是通过 `yield` 将生成器对象传入该生成器。

下面的 Node.js 脚本使用了这种技术，但是将生成器包裹在一个回调中（ `next` ，行 A ）。必须要通过 babel-node 运行。

```
import {readFile} from 'fs';

let fileNames = process.argv.slice(2);

run(function* () {
    let next = yield;
    for (let f of fileNames) {
        let contents = yield readFile(f, { encoding: 'utf8' }, n
ext);
        console.log('##### ' + f);
        console.log(contents);
    }
});
```

在行 A ，我们使用一个 Node.js 回调风格的的回调函数。该回调函数使用生成器对象唤醒生成器，你可以看一下 `run()` 的实现代码：

```
function run(generatorFunction) {
    let generatorObject = generatorFunction();

    // Step 1: Proceed to first `yield`
    generatorObject.next();

    // Step 2: Pass in a function that the generator can use as
a callback
    function nextFunction(error, result) { // (A)
        if (error) {
            generatorObject.throw(error);
        } else {
            generatorObject.next(result);
        }
    }
    generatorObject.next(nextFunction);

    // Subsequent invocations of `next()` are triggered by `next
Function`
}
```

21.6.3.3 通信顺序进程（ CSP ）

js-csp 将通信顺序进程（ CSP ）引入了 JavaScript 。 CSP 是一种多任务协作的方式，类似于 ClojureScript 的 core.async 和 Go 的 goroutine 。 `js-csp` 有两处抽象：

- 进程：是多任务协作的，通过将生成器函数传递给调度函数 `go()` 来实现。
- 通道：用于进程通信的队列。通过调用 `chan()` 创建通道。

作为一个示例，让我们使用 CSP 处理 DOM 事件，这是一种让人联想到函数式编程的方式。下面的代码使用函数 `listen()` （将在后面展示）创建一个输出 `mousemove` 事件的通道，然后在一个无限循环中通过 `take` 持续地获得输出内容。多亏了 `yield` ，进程才会阻塞直到通道有输出。

```
import csp from 'js-csp';

csp.go(function* () {
    let element = document.querySelector('#uiElement1');
    let channel = listen(element, 'mousemove');
    while (true) {
        let event = yield csp.take(channel);
        let x = event.layerX || event.clientX;
        let y = event.layerY || event.clientY;
        element.textContent = `${x}, ${y}`;
    }
});
```

`listen()` 的实现如下所示：

```
function listen(element, type) {
    let channel = csp.chan();
    element.addEventListener(type,
        event => {
            csp.putAsync(channel, event);
        });
    return channel;
}
```

> 此例来自 James Long 的博客《Taming the Asynchronous Beast with CSP Channels in JavaScript》。查阅这篇博客以获得更多关于 CSP 的信息。

21.7 基于迭代 API 的继承关系（包含生成器）

下面的图展示了在 ECMAScript 6 中各种对象之间的关系（此图基于 ECMAScript 规范中 Allen Wirf-Brock 的图）：



说明：

- 空心箭头表示两个对象的继承关系。换句话说，从 x 指向 y 的箭头意味着 `Object.getPrototypeOf(x) === y` 。
- 圆括号表示当前被包起来的对象是存在的，但是不能通过全局变量来访问。
- 带有 `instanceof` 字眼的箭头如果从 x 指向 y ，就表明 `x instanceof y` 。
  - `o instanceof C` 实际上就相当于 `C.prototype.isPrototypeOf(o)`
- 带有 `prototype` 字眼的箭头如果从 x 指向 y ，就表明 `x.prototype === y` 。

此图揭示了两个有趣的事实：

第一个，生成器函数 `g` 很像构造函数（甚至可以通过 new 来调用它，这和直接调用的效果是一样的）：它创建的生成器对象是它的实例，添加到 `g.prototype` 上的方法成为原型方法，等等：

```
> function* g() {}
> g.prototype.hello = function () { return 'hi!'};
> let obj = g();
> obj instanceof g
true
> obj.hello()
'hi!'
```

第二个，如果你想给所有的生成器对象添加方法，最好将这些方法添加到 `(Generator.object)` 上。访问这个对象的一种方式如下：

```
> let Generator_prototype = Object.getPrototypeOf(function* () {
}).prototype;
> Generator_prototype.hello = function () { return 'hi!'};
> let generatorObject = (function* () {})();
> generatorObject.hello()
'hi!'
```

21.7.1 IteratorPrototype

在图中没有 `(Iterator)` ，因为不存在这样的对象。但是，考虑到 `instanceof` 的工作原理，同时也因为 `(IteratorPrototype)` 是 `g1()` 的原型，你仍然可以说 `g1()` 是 `Iterator` 的实例。

ES6 中所有的迭代器在原型链中都有 `(IteratorPrototype)` 。这个对象是可迭代的，因为它有下面的方法。因此，所有的 ES6 迭代器都是可迭代的（因此，你可以在这些对象上应用 `for-of` ）。

```
[Symbol.iterator]() {
    return this;
}
```

规范推荐使用如下代码访问 `(IteratorPrototype)` ：

```
const proto = Object.getPrototypeOf.bind(Object);
let IteratorPrototype = proto(proto([][Symbol.iterator]()));
```

也可以使用：

```
let IteratorPrototype = proto(proto(function* () {}.prototype));
```

引用 ECMAScript 6 规范中的话：

> ECMAScript 代码可能定义继承自 IteratorPrototype 的对象。IteratorPrototype 对象提供了一个给所有可迭代对象添加 额外方法的地方。

在后续的 ECMAScript 版本中，也许可以直接获取到 IteratorPrototype ，并且包含一些工具方法，比如 `map()` 和 `filter()` （来源） 。

21.7.2 生成器中的 `this` 值

一个生成器函数混合了两方面的内容：

- 1、它是一个设置和返回生成器对象的函数。
- 2、它包含了生成器对象执行过程的代码。

这就是为什么生成器函数中的 `this` 值不太好猜测的原因。

在函数调用和方法调用中， `this` 值和其作为普通函数（非生成器函数）的时候一致：

```
function* gen() {
    'use strict'; // just in case
    yield this;
}

// Retrieve the yielded value via destructuring
let [functionThis] = gen();
console.log(functionThis); // undefined

let obj = { method: gen };
let [methodThis] = obj.method();
console.log(methodThis === obj); // true
```

如果通过 `new` 调用一个生成器函数，那么在函数中访问 this 将会得到一个 `ReferenceError` 错误（源自 ES6 规范）：

```
function* gen() {
    console.log(this); // ReferenceError
}
new gen();
```

有一个变通的方法，就是把生成器函数包裹在一个普通函数里面，通过 `next()` 传入生成器自身的生成器对象。这意味着生成器必须使用第一个 `yield` 拿到自身的生成器对象：

```
let generatorObject = yield;
```

21.8 代码风格：空格在星号之前还是之后

合理的并且合法的放置星号的方式是：

- 星号两边保留空格：

```
function * foo(x, y) { ··· }
```

- 星号之前保留空格：

```
function *foo(x, y) { ··· }
```

- 星号之后保留空格：

```
function* foo(x, y) { ··· }
```

- 星号前后都没有空格：

```
function*foo(x, y) { ··· }
```

让我们指出哪种形式对于哪种解构来说是合理的，并且解释为什么。

21.8.1 生成器函数的声明和表达式

此处，使用星号的原因是因为 `generator` （或者其它类似的单词）没有被用作关键字。如果可以作为关键字的话，那么生成器函数的声明就像这样子：

```
generator foo(x, y) {
    ...
}
```

ECMAScript 6 使用星号标记的 `function` 关键字来代替 `generator` 。因此，`function*` 可以看做是 `generator` 的同义词，这使得生成器函数声明的推荐写法就像下面这样。

```
function* foo(x, y) {
    ...
}
```

匿名生成器函数表达式的格式就像这样：

```
const foo = function* (x, y) {
    ...
}
```

21.8.2 生成器方法定义

当书写生成器方法定义的时候，我推荐如下的格式：

```
let obj = {
    * generatorMethod(x, y) {
        ...
    }
};
```

关于在星号之后放一个空格，有三个理由。

首先，星号不应该是方法名的一部分。一方面，它不是生成器函数名字的一部分。另一方面，仅在定义生成器的时候才会涉及到星号，在使用的时候并不会涉及到。

第二点，生成器方法定义是如下语法的缩写。（为了展现我的观点，我给函数表达式添加了多余的名字。）

```
let obj = {
    generatorMethod: function* generatorMethod(x, y) {
        ...
    }
};
```

如果方法定义要省略 `function` 关键字，那么星号后面就应该跟一个空格。

第三点，生成器方法定义在语法上类似于 `getters` 和 `setters` （这两个语法在 ECMAScript 5 中就可用了）：

```
let obj = {
    get foo() {
        ...
    }
    set foo(value) {
        ...
    }
};
```

关键字 `get` 和 `set` 可以看做普通方法定义的修饰器。同理，星号也是这样的一个修饰器。

21.8.3 用于嵌套调用的 yield 的书写格式

下面是一个 生成器函数通过 `yield` 递归调用自身的例子：

```
function* foo(x) {
    ...
    yield* foo(x - 1);
    ...
}
```

星号标记出不一样的 `yield` 操作符，这就是为什么上面的代码是合理的的原因。

21.8.4 标记生成器函数和方法

Kyle Simpson （ @getify ）提出了有趣的看法：在函数或方法调用的时候，我们经常会在后面加上括号，例如 `Math.max()` ，而对于生成器函数和方法而言，在前面加上星号不是更合理吗？例如：在上一子节我们不应该写成 `*foo()` 来表明是一个生成器的调用吗？让我来驳斥这个观点。

当需要调用一个返回可迭代对象的函数的时候，生成器是几个可选项中唯一的一个。我认为最好不要抛弃这种标记函数名的实现细节。

而且，在调用生成器函数的时候，你不会使用星号，但是使用了括号。

最后，星号并不会提供有用的信息 - `yield*` 也可用于返回可迭代对象的函数。但是标记返回可迭代对象的函数或方法名（包括生成器）可能是合理的。例如，加上 `Iter` 前缀。

21.9 总结

我希望本章使你相信生成器是有用的，拥有多种能力的工具。

我喜欢生成器实现多任务协作的方式：在异步调用的时候阻塞。我认为这是正确的异步调用方式。希望未来 JavaScript 在这个方向上越走越远。

21.10 深入阅读

本章来源：

- [1] 《Async Generator Proposal》，作者 Jafar Husain
- [2] 《A Curious Course on Coroutines and Concurrency》，作者 David Beazley
- [3] 《Why coroutines won't work on the web》，作者 David Herman

24 用于异步编程的 Promise

本章介绍了用 Promise 做异步编程的一般知识点，着重介绍了 ECMAScript 6 的 Promise API 。上一章讲解了 JavaScript 中异步编程的基础知识。在本章中遇到不理解的东西，可以参考上一章内容。

24.1 概览

下面的函数通过 Promise 异步地返回结果：

```
function asyncFunc() {
    return new Promise(
        function (resolve, reject) {
            resolve(value); // success
            ...
            reject(error); // failure
        });
}
```

像下面这样调用 `asyncFunc()` ：

```
asyncFunc()
.then(value => { /* success */ })
.catch(error => { /* failure */ });
```

24.1.1 处理 Promise 数组

`Promise.all()` 使你能够应对 Promise 数组。

例如，可以通过 Array 的方法 `map()` 创建一个 Promise 数组：

```
let fileUrls = [
    'http://example.com/file1.txt',
    'http://example.com/file2.txt'
];
let promisedTexts = fileUrls.map(httpGet); // Array of Promises
```

如果对这个数组调用 `Promise.all()` ，那么在所有 Promise 成功完成之后，会得到一个值数组：

```
Promise.all(promisedTexts)
// Success
.then(texts => {
    for (let text of texts) {
        console.log(text);
    }
})
// Failure
.catch(reason => {
    // Receives first rejection among `promisedTexts`
});
```

24.2 Promise

Promise 是一种帮助处理一类特定的异步编程的方式：一个函数（或者方法）异步地返回结果。为了实现这样的函数，需要返回一个 Promise ，它是最终返回结果的一个占位符对象。函数的调用者在 Promise 对象上注册回调，一旦结果计算出来了，就可以收到通知。函数通过 Promise 发送结果。

JavaScript Promise 的实际标准叫做 Promise/A+。 ECMAScript 6 的 Promise API 遵循这个标准。

24.3 第一个例子

让我们看下第一个例子，初步了解使用 Promise 会是怎么样的。

在 Node.js 风格的回到中，异步地读取一个文件的内容看起来像这样：

```
fs.readFile('config.json',
    function (error, text) {
        if (error) {
            console.error('Error while reading config file');
        } else {
            try {
                let obj = JSON.parse(text);
                console.log(JSON.stringify(obj, null, 4));
            } catch (e) {
                console.error('Invalid JSON in file');
            }
        }
    });
```

使用 Promise ，相同功能的实现像这样：

```
readFilePromisified('config.json')
.then(function (text) { // (A)
    let obj = JSON.parse(text);
    console.log(JSON.stringify(obj, null, 4));
})
.catch(function (reason) { // (B)
    // File read error or JSON SyntaxError
    console.error('An error occurred', reason);
});
```

仍然有回调，但是这些回调通过函数返回结果（ Promise ）上的方法（ `then()` 和 `catch()` ）来提供。行 B 的错误回调在两个方面是比较方便的：首先，仅用于处理错误（对比一下之前例子中的 `if(error)` 和 `try-catch` ）；其次，可以在某一个地方处理 `readFilePromisified()` 和行 A 处的回调产生的错误。

`readFilePromisified()` 的代码在后面展示。

24.4 创建和使用 Promise

让我们看一下生产端和消费端是如何操作 Promise 的。

24.4.1 生成 Promise

作为一个生产者，需要创建一个 Promise ，然后通过它发送最终结果：

```
let promise = new Promise(
    function (resolve, reject) { // (A)
        ...
        if (···) {
            resolve(value); // success
        } else {
            reject(reason); // failure
        }
    });
```

一个 Promise 总是处于三种状态（互相排斥）中的一种：

- 等待（ pending ）：结果还没计算出来
- 成功完成（ fullfilled ）：结果成功地被计算出来了
- 被驳回（ rejected ）：在计算的过程中失败了

如果一个 Promise 已经成功完成或者被驳回了，那么它就会保持这个状态不再变化了。一个 Promise 只能有一次机会变稳定，然后就一直保持稳定了。接下来试图使它变稳定的操作都不会有效果。



`new Promise()` （开始于行 A ）的参数被称为执行者（ executor ）：

- 如果计算过程顺利执行，执行者通过 `resolve()` 发送执行结果。这通常会将 Promise 置为成功完成的状态（如果返回一个 Promise ，可能就不会这样，就像后面讲解的）。
- 如果出错了，执行者通过 `reject()` 通知 Promise 。这总是会将 Promise 置为被驳回状态。

24.4.2 消费 Promise

作为一个 Promise 的消费者，会在通过 `then()` 注册的回调中接收到成功完成或者被驳回的通知：

```
promise.then(
    function (value) { /* fulfillment */ },
    function (reason) { /* rejection */ }
);
```

对于异步函数（最终结果是一次性的），让 Promise 如此有用的原因是，一旦 Promise 稳定了，就不会变化了。而且，没有任何竞态条件，因为在 Promise 稳定之前或之后调用 `then()` 方法是无关紧要的：

- 在 Promise 稳定之前注册的回调，一旦 Promise 变为稳定状态，就会收到通知。
- 在 Promise 稳定之后注册的回调，会“立即”接收到缓存下来的稳定状态值（回调函数像任务队列一样被调用）。

24.4.3 仅处理成功完成或被驳回

如果仅对成功完成感兴趣，可以忽略 `then()` 的第二个参数：

```
promise.then(
    function (value) { /* fulfillment */ }
);
```

如果仅对被驳回感兴趣，可以省略第一个参数。`catch()` 方法是一种完成相同功能的更紧凑方式。

```
promise.then(
    null,
    function (reason) { /* rejection */ }
);

// Equivalent:
promise.catch(
    function (reason) { /* rejection */ }
);
```

推荐用 `then()` 处理成功完成，`catch()` 处理错误，因为这样就给回调打上了贴切的标签，也因为可以同时处理多个 Promise 的驳回（后面详细讲解）。

24.5 例子

在我们更加深入了解 Promise 之前，让我们在几个例子里面使用已经学到的知识。

> 本节的一些例子可以在 GitHub 仓库 promise-examples 中找到。

24.5.1 例子：Promise 化 `fs.readFile()`

下面的代码是一个内置的基于 Promise 版本的 Node.js 函数 fs.readFile() 。

```
import {readFile} from 'fs';

function readFilePromisified(filename) {
    return new Promise(
        function (resolve, reject) {
            readFile(filename, { encoding: 'utf8' },
                (error, data) => {
                    if (error) {
                        reject(error);
                    }
                    resolve(data);
                });
        });
}
```

`readFilePromisified()` 像下面这样使用：

```
readFilePromisified(process.argv[2])
.then(text => {
    console.log(text);
})
.catch(error => {
    console.log(error);
});
```

24.5.2 例子：Promise 化 XMLHttpRequest

下面是一个基于 Promise 的函数，通过基于事件的 XMLHttpRequest API 执行 HTTP GET 请求：

```
function httpGet(url) {
    return new Promise(
        function (resolve, reject) {
            let request = new XMLHttpRequest();
            request.onreadystatechange = function () {
                if (this.status === 200) {
                    // Success
                    resolve(this.response);
                } else {
                    // Something went wrong (404 etc.)
                    reject(new Error(this.statusText));
                }
            }
            request.onerror = function () {
                reject(new Error(
                    'XMLHttpRequest Error: '+this.statusText));
            };
            request.open('GET', url);
            request.send();
        });
}
```

下面是使用 `httpGet()` 的方式：

```
httpGet('http://example.com/file.txt')
.then(
    function (value) {
        console.log('Contents: ' + value);
    },
    function (reason) {
        console.error('Something went wrong', reason);
    });
```

24.5.3 例子：延迟操作

让我们将 `setTimeout()` 实现为基于 Promise 的函数 `delay()` （类似于 Q.delay()）。

```
function delay(ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, ms); // (A)
    });
}

// Using delay():
delay(5000).then(function () { // (B)
    console.log('5 seconds have passed!')
});
```

注意，在行 A ，调用 `resolve` 的时候没传参数，这和 `resolve(undefined)` 是一样的。在行 B 也不需要成功完成的结果值，所以简单地忽略它。这里仅仅接收通知就够了。

24.5.4 例子： Promise 超时

```
function timeout(ms, promise) {
    return new Promise(function (resolve, reject) {
        promise.then(resolve);
        setTimeout(function () {
            reject(new Error('Timeout after '+ms+' ms')); // (A)
        }, ms);
    });
}
```

注意，在超时之后的驳回（行 A ）并不会取消请求，

24.6 链式使用 Promise

现在已经准备好深入 Promise 特性了。首先探索一下如何链式使用 Promsie 。

方法调用的结果：

```
P.then(onFulfilled, onRejected)
```

是一个新的 Promise Q 。这意味着你可以通过在 Q 上调用 `then()` 来维持基于 Promise 的控制流：

- 如果 `onFulfilled` 或 `onRejected` 成功返回，则 Q 变为 fulfilled 状态，相应的结果值就是前面函数的返回值。
- 如果 `onFulfilled` 或 `onRejected` 抛出异常，则 Q 变为 rejected 状态。

24.6.1 用普通的值将 Q 置为 fulfilled 状态

如果用一个普通的值将 `then()` 返回的 Promise Q 置为 fulfilled 状态，可以追加一个 `then()` 来得到那个值：

```
asyncFunc()
.then(function (value1) {
    return 123;
})
.then(function (value2) {
    console.log(value2); // 123
});
```

24.6.2 用带有 then 方法的对象将 Q 置为 fulfilled 状态

也可以用带有 then 方法的对象 R 将 `then()` 返回的 Promise Q 置为 fulfilled 状态。可 then 化（ thenable ）实体就是任何一个有 Promise 风格方法 `then()` 的对象。因此， Promise 是可 then 化的。用 R 置为 fulfilled 状态（例如在 `onFulfilled` 中返回 R ）意味着 R 被插在 Q "后面"： R 的稳定消息会转发给 Q 的 `onFulfilled` 和 `onRejected` 回调。在某种程度上， Q 变成了 R 。



这种机制的主要用途就是扁平化内嵌的 `then()` 调用，就像下面的例子：

```
asyncFunc1()
.then(function (value1) {
    asyncFunc2()
    .then(function (value2) {
        ...
    });
})
```

扁平化的版本看起来像这样：

```
asyncFunc1()
.then(function (value1) {
    return asyncFunc2();
})
.then(function (value2) {
    ...
})
```

24.6.3 从 `onRejected` 将 Q 置为 fulfilled 状态

在错误处理器中返回的内容变成了一个成功完成的值（而不是被驳回的值）。这使得可以在执行失败的时候指定默认值：

```
retrieveFileName()
.catch(function () {
    // Something went wrong, use a default value
    return 'Untitled.txt';
})
.then(function (fileName) {
    ...
});
```

24.6.4 通过抛出异常将 Q 置为 rejected 状态

在 `then()` 的两个回调参数中抛出的异常会传递给下一个错误处理器：

```
asyncFunc()
.then(function (value) {
    throw new Error();
})
.catch(function (reason) {
    // Handle error here
});
```

24.6.5 执行者中的异常

在执行者（ `new Promise()` 的回调）中抛出的异常传递给该执行者负责管理的 Promise ：

```
new Promise(function (resolve, reject) {
    throw new Error();
})
.catch(function (err) {
    // Handle error here
});
```

24.6.6 链式错误

可以有一个或多个 `then()` 方法调用，并且这些方法调用中都没有传递错误处理器。此时错误就会一直往下传递，直到遇见一个错误处理器。

```
asyncFunc1()
.then(asyncFunc2)
.then(asyncFunc3)
.catch(function (reason) {
    // Something went wrong above
});
```

24.7 组合

本节讲述了如何将已有的 Promise 组合起来创建新的 Promise 。我们已经遇到了一种组合 Promise 的方法：通过 `then()` 依次链式连接。 `Promise.all()` 和 `Promise.race()` 提供了另外的组合方式。

24.7.1 `map()` 与 `Promise.all()` 结合

关于 Promise 一件很好的事情就是很多同步的工具仍然有效，因为基于 Promise 的函数有返回结果。例如，可以使用数组方法 `map()` ：

```
let fileUrls = [
    'http://example.com/file1.txt',
    'http://example.com/file2.txt'
];
let promisedTexts = fileUrls.map(httpGet);
```

`promisedTexts` 是一组 Promise 。 `Promise.all()` 用一个 Promise 数组（可 then 化的和其它值都通过 `Promise.resolve()` 被转换成了 Promise ）作为参数，一旦所有的 Promise 都成功完成了，它就变为 fulfilled 状态，返回结果是这组 Promise 的返回结果的一个数组：

```
Promise.all(promisedTexts)
.then(texts => {
    for (let text of texts) {
        console.log(text);
    }
})
.catch(reason => {
    // Receives first rejection among the Promises
});
```

24.7.2 利用 `Promise.race()` 实现超时

`Promise.race()` 使用一个 Promise 数组（可 then 化的和其它值都通过 `Promise.resolve()` 被转换成了 Promise ）作为参数，然后返回一个 Promise P 。传入的 Promise 数组中只要有一个 Promise 变为稳定状态，就会将稳定状态信息传递给返回的那个 Promise P 。

让我们使用 `Promise.race()` 实现超时：

```
Promise.race([
    httpGet('http://example.com/file.txt'),
    delay(5000).then(function () {
        throw new Error('Timed out')
    });
])
.then(function (text) { ··· })
.catch(function (reason) { ··· });
```

24.8 Promise 总是异步的

一个 Promise 库完全可以决定计算结果是同步地（立即）传递给 Promise 接应器，还是异步地（在当前代码片段执行完之后）。但是， Promises/A+ 规范要求总是使用后一种执行模式。规范通过下列用于 `then()` 方法的必要条件陈述了这个问题：

> `onFulfilled` 或 `onRejected` 一定不能被调用，直到执行上下文堆栈仅包含平台代码。

这意味着你的代码可以依赖于代码执行的完整性语义（ run-to-completion semantics ）（如同第一部分讲解的），并且这种链式的 Promise 并不会阻碍其它任务的执行。

24.9 备忘单： ECMAScript 6 Promise API

本节给出 ECMAScript 6 Promise API 的一个概览，和规范中描述的一样。

24.9.1 Promise 相关术语

Promise API 是关于异步返回结果的。一个 Promise 对象（简称 Promise ）就是一个结果的替身，结果将会通过这个对象返回。

状态：

- 一个 Promise 总是处于三个互相排斥状态中的一种：
    - 在结果就绪之前， Promise 变为“未完成（ pending ）”状态。
    - 计算结果就绪， Promise 变为“已完成（ resolved 或 fulfilled ）”状态。
    - 如果发生了错误， Promise 变为“已失败（ rejected ）”状态。
- 如果“所有事情都完成了”（也就是说要么完成要么失败），那么 Promise 就稳定了。
- Promise 仅会稳定一次，然后就保持不变了。

状态改变的回调函数：

- Promise 回调函数就是利用 Promise 的 `then()` 方法注册的那些函数，在 Promise 完成或失败的时候会调用相应的回调函数。
- 一个 thenable 实体是一个有 Promise 风格的 `then()` 方法的对象。 API 只关心在 Promise 稳定的时候能够收到通知，因此只需要 thenable 实体。

改变状态：有两种操作可以改变 Promise 的状态。在执行了一次两种操作中的一种之后，将来再执行这两种操作中的任何一种都没有效果了。

- Promise 失败意味着 Promise 变为了已失败状态。
- Promise 完成有不同的效果，依赖于完成的值：
    - 完成的值是一个普通的（非 thenable 的）值，会使当前 Promise 变为已完成。
    - Promise P 完成之后得到一个 thenable 的 T ，意味着 P 不能再变为已完成状态，并且后面将会追踪 T 的状态，包括 T 的已完成和已失败的值。一旦 T 稳定了，相应的 P 的回调函数就会被调用（或者，如果已经稳定，则会立马调用）。

24.9.2 `Promise` 构造器

Promise 构造器像下面这样调用：

```
let p = new Promise(function (resolve, reject) { ··· });
```

这个构造器的回电函数被称作执行者（ executor ）。执行者可以使用它的参数来使新的 Promise `p` 完成或失败。

- `resolve(x)` 使 `p` 完成，并返回 `x` 作为结果：
  - 如果 `x` 是 thenable 的，它的稳定消息就会转发给 `p` （这会触发通过 `then()` 注册的回调函数）。
  - 否则， `p` 变成已完成状态，返回的结果是 `x` 。

24.9.3 静态的 `Promise` 方法

`Promise` 的所有静态方法支持子类化：

```
* 它们通过所谓的种模式（这在关于类的那一章有详细讲解）创建新的 Promise 实例
。
    * 默认情况下使用接收器（ this ）作为构造器。
    * 默认值可以通过在子类中重写 `[Symbol.species]` 来覆盖。
* 其它静态方法也通过种模式访问（而不是通过 `Promise` 或者 `this` ）。
```

24.9.3.1 创建 Promise

下面的两个静态方法创建它们的新的接收器实例：

- `Promise.resolve(x)` ：将任何值装换为 Promise 对象，使其和 thenable 、 Promise 表现一致。
  - 如果 `x` 是 thenable 的，就会被转换成 Promise - 一个接收器实例（ this ；此处未使用种模式）。
  - 如果 `x` 是接收器实例，该实例会被直接返回。
  - 否则，返回接收器的新实例，该实例处于完成状态，返回值是 `x` 。
- `Promise.reject(reason)` ：创建一个新的接收器实例（就像种模式中配置的一样），该实例处于失败状态，返回值是 `reason` 。

24.9.3.2 组合 Promise

从表面上看，静态方法 `Promise.all()` 和 `Promise.race()` 将一组可迭代的 Promise 转变为一个单一的 Promise 。也就是说：

- 可迭代。迭代器的元素通过 `this.resolve()` 转换为 Promise 。
- 返回一个新的 Promise 对象。该 Promise 对象是一个新的接收器实例（正如通过种模式配置的一样）。

方法是：

- `Promise.all(iterable)` ：返回一个 Promise ...
  - 如果迭代器中所有元素都处于已完成状态，则返回的 Promise 对象就处于已完成状态。已完成的值：已完成值的数组。
  - 如果有任何一个元素处于已失败状态，则返回的 Promise 对象就处于已失败状态。已失败的值：第一个失败元素的返回值。

- `Promise.race(iterable)` ：迭代器中只要有一个元素处于稳定状态，那么返回的 Promise 就会立马变为一样的稳定状态。

24.9.4 `Promise.prototype` 上的方法

24.9.4.1 `Promise.prototype.then(onFulfilled, onRejected)`

```
* onFulfilled 和 onRejected 回调函数被称为 reactions 。
* 如果 Promise 已完成或者只要 Promise 变成已完成状态，就会立刻调用 onFul
filled 。类似地，在失败的时候也会调用 `onRejected` 。
* `then()` 返回一个新的 Promise Q （ 通过接收器的构造函数种类 ）：
    * 如果两个回调函数中的一个返回值， Q 以该值变为 resolved 状态。
    * 如果两个回调函数中的一个抛出异常， Q 以该异常变为 rejected 状态。
* 省略回调函数：
    * 如果省略 `onFulfilled` ，那么 fulfilled 状态就会转发给 `then()`
 的结果。
    * 如果省略 `onRejected` ，那么 rejected 状态就会转发给 `then()`
的结果。
```

回调函数的默认实现可能像这样：

```
function defaultOnFulfilled(x) {
    return x;
}
function defaultOnRejected(e) {
    throw e;
}
```

24.9.4.2 `Promise.prototype.catch(onRejected)`

- `p.catch(onRejected)` 等价于 `p.then(null, onRejected)` 。

24.10 Promise 的优缺点

24.10.1 优点

24.10.1.1 统一异步 API

Promise 的一个重要优点是它将逐渐被用作浏览器的异步 API ，统一现在各种各样的 API ，以及不兼容的模式和手法。让我们看两个即将到来的基于 Promise 的 API 。

fetch API 是基于 Promise 的，用于处理 XMLHttpRequest 的一种方式：

```
fetch(url)
.then(request => request.text())
.then(str => ···)
```

在实际请求中， `fetch()` 返回一个 Promise ； `text()` 也返回一个 Promise ，用于将响应内容转换成字符串。

ES6 中可编程地动态引入模块也是基于 Promise 的：

```
System.import('some_module.js')
.then(some_module => {
    ···
})
```

24.10.1.2 Promise 与事件对比

和事件相比较， Promise 更适合处理一次性的结果。在结果计算出来之前或之后注册回调函数都是可以的，都可以拿到正确的值。 Promise 的这个优点很自然。但是，不能使用 Promise 处理多次触发的事件。链式处理是 Promise 的又一优点，但是事件却不能这样链式处理。

24.10.1.3 Promise 与回调对比

和回调函数比较， Promise 有更干净的函数（或者方法）签名。回调函数的场景，主函数既有输入参数，又有输出参数：

```
// name 和 opt 是输入参数， (err, string | Buffer) => void 是输出参数

fs.readFile(name, opts?, (err, string | Buffer) => void)
```

Promise 的场景，所有的参数都是输入参数：

```
readFilePromisified(name, opts?) : Promise<string | Buffer>
```

Promise 带来的额外好处是包含了更好的错误处理方式（包含了异常处理），并且写起来很轻松（因为可以重用一些同步的工具，比如 `Array.prototype.map()` ）。

24.10.2 缺点

Promise 可以很好地处理单一异步结果，不适用于：

- 多次触发的事件：如果要处理这种情况，可以了解一下响应式编程（ reactive programming ），这是一种很聪明的链式的处理普通事件的方法。
- 数据流：支持此种情形的标准正在制定中。

ECMAScript 6 Promise 缺少两个有时很有用的特性：

- 不能取消执行。
- 无法获取当前执行的进度信息（比如，要在用户界面展示进度条）。

Q Promise 库对于后者提供了支持，并且有计划将两种能力都添加到 `Promises/A+` 规范中去。

24.11 Promise 和 generator

在下面的代码中，我使用流程控制库 co 来异步读取两个 JSON 文件。注意在行 A ，执行块会一直等到 `Promise.all()` 处理完成。这意味着看起来同步的代码实际上执行了异步的操作。

```
co(function* () {
    try {
        let [croftStr, bondStr] = yield Promise.all([  // (A)
            getFile('http://localhost:8000/croft.json'),
            getFile('http://localhost:8000/bond.json'),
        ]);
        let croftJson = JSON.parse(croftStr);
        let bondJson = JSON.parse(bondStr);

        console.log(croftJson);
        console.log(bondJson);
    } catch (e) {
        console.log('Failure to read: ' + e);
    }
});
```

详细内容在 21.5.2.1 节讲解。

24.12 调试 Promise

调试 Promise 的工具逐渐在浏览器中出现了。让我们看下最新的谷歌浏览器提供了什么。下面是一个 HTML 文件的一部分，展示了使用 Promise 的两个常见问题：

```
<body>
    <script>
        // Unhandled rejection
        Promise.reject(new Error())
        .then(function (x) { return 'a'})
        .then(function (x) { return 'b'})

        // Unsettled Promise
        new Promise(function () {});
    </script>
</body>
```

第一段 JavaScript 代码中，没有处理异常情况。第二段 JavaScript 代码中，Promise 没有变为稳定状态。谷歌浏览器的开发者工具会帮助处理这两种问题。

未处理的异常将会被打印在控制台里：



同时也会在监视器的“ source ”标签页下高亮：



最后，有一个特殊的用于 Promise 的监视器会记录 web 页面创建的所有 Promise。除了记录其它东西，还会记录：

- 哪些 Promise 完成了（在第一列中显示绿色圆点），哪些失败了（红色圆点），哪些还在执行中（灰色圆点）
- Promise 是如何连接的

24.13 Promise 内部机制

本节中，我们将会从一个不一样的角度来学习 Promise ：我们将会简单实现 Promise ，而不是学习如何使用 Promise API 。这种不同的视角很大程度地帮助我理解了 Promise 。

此处 Promise 的实现叫做 `DemoPromise` 。为了理解起来更容易，并没有完全符合 ES6 的 Promise API 。但是仍然足以说明实际的 Promise 实现会遇到什么挑战。

> `DemoPromise` 可以在 GirHub 上找到，位于仓库 demo_promise 中。

`DemoPromise` 是有三个原型方法的类：

- `DemoPromise.prototype.resolve(value)`
- `DemoPromise.prototype.reject(reason)`
- `DemoPromise.prototype.then(onFulfilled, onRejected)`

也就是说， `resolve` 和 `reject` 是方法（比较传递给构造函数 Promise 的回调函数的参数）。

24.13.1 一个独立的 Promise

第一个实现是一个独立的 Promise ，只有最少的功能：

- 可以创建 Promise
- 可以通过（ resolve ）或者拒绝（ reject ）一个 Promise ，并且只能做一次。
- 可以通过 `then()` 注册应答器（回调函数）。这一定要与 Promise 是否已经稳定无关。
    - 该方法暂不支持链式使用 - 不反悔任何东西。

下面是如何使用第一个版本的 Promise ：

```
let dp = new DemoPromise();
dp.resolve('abc');
dp.then(function (value) {
    console.log(value); // abc
});
```

下图说明了我们的第一个 `DemoPromise` 是如何工作的：



24.13.1.1 `DemoPromise.prototype.then()`

首先检查 `then()` 。该方法处理两种情形：

- 如果 Promise 还处于 `pending` 状态，则会将 `onFulfilled` 和 `onRejected` 的调用存储在队列中，当 Promise 稳定下来的时候就使用。
- 如果 Promise 已经处于 fulfilled 状态或者 rejected 状态， `onFulfilled` 或者 `onRejected` 可能会被立刻调用。

```
then(onFulfilled, onRejected) {
    let self = this;
    let fulfilledTask = function () {
        onFulfilled(self.promiseResult);
    };
    let rejectedTask = function () {
        onRejected(self.promiseResult);
    };
    switch (this.promiseState) {
        case 'pending':
            this.fulfillReactions.push(fulfilledTask);
            this.rejectReactions.push(rejectedTask);
            break;
        case 'fulfilled':
            addToTaskQueue(fulfilledTask);
            break;
        case 'rejected':
            addToTaskQueue(rejectedTask);
            break;
    }
}
```

上面的代码片段使用了下面的辅助函数：

```
function addToTaskQueue(task) {
    setTimeout(task, 0);
}
```

24.13.1.2 `DemoPromise.prototype.resolve()`

`resolve()` 像下面这样工作：如果 Promise 已经稳定了，什么事也不做（确保 Promise 只能变为稳定状态一次）。否则， Promise 的状态变为 `fulfilled` ，结果缓存在 `this.promiseResult` 中。接下来，触发所有之前存放的 fulfilled 状态相关的回调函数。

```
resolve(value) {
    if (this.promiseState !== 'pending') return;
    this.promiseState = 'fulfilled';
    this.promiseResult = value;
    this._clearAndEnqueueReactions(this.fulfillReactions);
    return this; // enable chaining
}
_clearAndEnqueueReactions(reactions) {
    this.fulfillReactions = undefined;
    this.rejectReactions = undefined;
    reactions.map(addToTaskQueue);
}
```

`reject()` 类似于 `resolve()` 。

24.13.2 链式

接下来要实现的特性是链式：

- `then()` 返回一个 Promise ，它通过 `onFulfilled` 或 `onRejected` 的返回来变成 `resolved` 状态。
- 如果省略 `onFulfiled` 或 `onRejected` ，那么不管接收到什么，都会传给 `then()` 返回的 Promise 。



很明显，只需要修改 `then()` ：

```
then(onFulfilled, onRejected) {
    let returnValue = new Promise(); // (A)
    let self = this;

    let fulfilledTask;
    if (typeof onFulfilled === 'function') {
        fulfilledTask = function () {
            let r = onFulfilled(self.promiseResult);
            returnValue.resolve(r); // (B)
        };
    } else {
        fulfilledTask = function () {
            returnValue.resolve(self.promiseResult); // (C)
        };
    }

    let rejectedTask;
    if (typeof onRejected === 'function') {
        rejectedTask = function () {
            let r = onRejected(self.promiseResult);
            returnValue.resolve(r); // (D)
        };
    } else {
        rejectedTask = function () {
            // `onRejected` has not been provided
            // => we must pass on the rejection
            returnValue.reject(self.promiseResult); // (E)
        };
    }
    ...
    return returnValue; // (F)
}
```

`then()` 创建并返回了一个新的 Promise （行 A 和 F）， `fulfilledTask` 和 `rejectedTask` 被分别设置：在稳定之后......

- `onFulfilled` 的返回值传入 `returnValue` 的 resolve 方法（行 B ）。
  - 如果没有传入 `onFulfilled` ，就将当前已经完成的值传入 `returnValue` 的 resolve 方法（行 C ）。
- `onRejected` 的返回值传入 `returnValue` 的 resolve 方法（不是 reject 方法！）（行 D ）。
  - 如果没有传入 `onRejected` ，就将当前已经完成的值传入 `returnValue` 的 reject 方法（行 E ）。

24.13.3 扁平化（ Flattening ）

扁平化主要让链式调用更加方便：通常情况下，将响应器（ reaction ）返回的值传给下一个 `then()` 。如果返回一个 Promise ，并且内部并没有做相应的“包装”，这种情形是非常美好的，就像下面的例子：

```
asyncFunc1()
.then(function (value1) {
    return asyncFunc2(); // (A)
})
.then(function (value2) {
    // value2 is fulfillment value of asyncFunc2() Promise
    console.log(value2);
});
```

在行 A 返回一个 Promise ，没必要将下一个 `then()` 内嵌到当前方法中，可以在当前方法的返回值上调用 `then()` 。所以：没有内嵌的 `then()` ，所有内容保持扁平。

通过 `resolve()` 方法实现这种扁平功能：

- 用 Promise Q 来解析 Promise P 意味着 Q 的稳定情况会转发给 P 。
- P 被“锁在了”Q 上：不能再手动变为 resolved 状态（同样也不能变为 rejected 状态）。它的状态和结果总是和 Q 保持一致。

如果让 Q 变成 thenable 对象（而不仅仅是一个 Promise ），那么就可以让扁平化更加通用，



为了实现锁定，引入一个新的布尔标志 `this.alreadyResolved` 。一旦该值变为 `true` ， `this` 就被锁定了，不能再被解析（ resolve ）了。注意此时 `this` 可能还处于 `pending` 状态，因为它的状态现在和锁定到的 Promise 一致。

```
resolve(value) {
    if (this.alreadyResolved) return;
    this.alreadyResolved = true;
    this._doResolve(value);
    return this; // enable chaining
}
```

实际的解析过程现在发生在私有的方法 `_doResolve()` 中：

```
_doResolve(value) {
    let self = this;
    // Is `value` a thenable?
    if (typeof value === 'object' && value !== null && 'then' in
 value) {
        // Forward fulfillments and rejections from `value` to `
this`.
        // Added as a task (vs. done immediately) to preserve as
ync semantics.
        addToTaskQueue(function () { // (A)
            value.then(
                function onFulfilled(result) {
                    self._doResolve(result);
                },
                function onRejected(error) {
                    self._doReject(error);
                });
        });
    } else {
        this.promiseState = 'fulfilled';
        this.promiseResult = value;
        this._clearAndEnqueueReactions(this.fulfillReactions);
    }
}
```

在行 A 处理了扁平化：如果 `value` 是 fulfilled 状态，我们希望 `self` 也是 fulfilled 状态；如果 `value` 是 rejected 状态，我们希望 `self` 也是 rejected 状态。通过私有方法 `_doResolve` 和 `_doReject` 不停转发，绕过 `alreadyResolved` 标志。

24.13.4 Promise 状态的更多细节

实现链式特性之后， Promise 的状态更加复杂了（正如 ECMAScript 6 规范的25.4节描述的那样）：



如果你仅使用 Promise ，通常你可以采用一种简单的方式，忽略掉锁定。最重要的状态相关的概念仍然是“稳定（ settledness ）”：一个 Promise 在变为 fulfilled 状态或 rejected 状态，就稳定了。在 Promise 稳定之后，就不再会变化了（状态和最终的值都不会变化）。

如果你想实现 Promise ，那么“解析（ resolving ）”也很重要，并且现在很难理解：

- 直观地，“ resolved ”的意思是“不能再被（直接地）解析了”。如果一个 Promise 稳定了或者锁定了，那么就变成了 `resolved` 状态。引用规范里面的话：“一个未解析（ unresolved ）的 Promise 总是处于 pending 状态。一个解析（ resolved ）的 Promise 可能是 pending 状态，也可能是 fulfilled 或

rejected 状态”。
- 解析（ resolving ）不一定会使 Promise 变为稳定状态：你可以用另外一个总是处于 pending 状态的 Promise 来解析某个 Promise 。
- 解析（ resolving ）现在包括了拒绝（ rejecting ）（即更一般化了）：你可以通过一个处于 rejected 状态的 Promise 来解析某个 Promise ，从而使其也变为 rejected 状态。

24.13.5 异常

最后一个特性，我们希望在用户代码抛出异常的时候进入拒绝（ rejection ）回调函数。此处，“用户代码”指的是 `then()` 的两个回调参数。



下面的代码展示了我们如何将 `onFulfilled` 中抛出的异常转移到拒绝回调函数中 - 通过在行 A 用 `try-catch` 包裹 `onFulfilled` 的调用。

```
then(onFulfilled, onRejected) {
    ...
    let fulfilledTask;
    if (typeof onFulfilled === 'function') {
        fulfilledTask = function () {
            try {
                let r = onFulfilled(self.promiseResult); // (A)
                returnValue.resolve(r);
            } catch (e) {
                returnValue.reject(e);
            }
        };
    } else {
        fulfilledTask = function () {
            returnValue.resolve(self.promiseResult);
        };
    }
    ...
}
```

24.13.6 启发性的构造器模式

如果我们想将 `DemoPromise` 变成一个实际可用的 Promise 实现，还需要实现启发性的构造器模式[5]： ES6 的 Promise 不通过方法来解析（ resolve ）和拒绝（ reject ），而是通过传递给执行器（ executor ）的函数，即构造器的回调参数。



如果执行器抛出异常，那么对应的 Promise 就会变为 rejected 状态。

24.14 两个有用的 Promise 附加方法

本节介绍两个很有用的方法，这两个方法可以很容易的添加到 ES6 的 Promise 上面去。很多更全面的 Promise 库都有这两个方法。

24.14.1 `done()`

当你链接若干个 Promise 方法调用的时候，有一不小心忘记处理某些错误的风险。例如：

```
function doSomething() {
    asyncFunc()
    .then(f1)
    .catch(r1)
    .then(f2); // (A)
}
```

如果在行 A 的 `then()` 触发了拒绝（ rejection ），那么这个错误将不会被处理。Promise 库 Q 提供了一个方法 `done()` ，被用作链式调用的最后一个调用元素。`done()` 要么替换掉最后一个 `then()` （此时有一到两个参数）：

```
function doSomething() {
    asyncFunc()
    .then(f1)
    .catch(r1)
    .done(f2);
}
```

要么放在最后一个 `then()` 的后面（此时没有参数）：

```
function doSomething() {
    asyncFunc()
    .then(f1)
    .catch(r1)
    .then(f2)
    .done();
}
```

引用 Q 的文档：

> `done` 和 `then` 使用的黄金规则就是：要么返回 Promise 给其它地方，要么调用 `done` 结束当前 Promise 链（如果这条链在当前位置结束的话）。用 `catch` 来结束 Promise 是不完美的，因为 catch 的处理器自身可能会抛出错误。

下面就是在 ECMAScript 6 中实现 `done()` 的方式：

```
Promise.prototype.done = function (onFulfilled, onRejected) {
    this.then(onFulfilled, onRejected)
    .catch(function (reason) {
        // Throw an exception globally
        setTimeout(() => { throw reason }, 0);
    });
};
```

虽然 `done` 的功能很有用，但是还是没有添加到 ECMAScript 6 中去，因为这种检查可以通过引擎自动实现（正如在调试 Promise 的那一节看到的一样）。

24.14.2 `finally()`

有时你想执行一些操作而不管是否发生了错误。例如，在操作完一个资源之后做一些清理工作。这就是 Promise 方法 `finally` 适用的场景，它和异常处理中的 finally 子句很相似。它的回调函数不接收参数，但是不管是 rejected 还是 resolved 状态都会被调用。

```
createResource(···)
.then(function (value1) {
    // Use resource
})
.then(function (value2) {
    // Use resource
})
.finally(function () {
    // Clean up
});
```

下面是 `Domenic Denicola` 提议的实现 `finally()` 的方式：

```
Promise.prototype.finally = function (callback) {
    let P = this.constructor;
    // We don’t invoke the callback in here,
    // because we want then() to handle its exceptions
    return this.then(
        // Callback fulfills => continue with receiver’s fulfill
ment or rejection
        // Callback rejects => pass on that rejection (then() ha
s no 2nd paramet\
er!)
        value  => P.resolve(callback()).then(() => value),
        reason => P.resolve(callback()).then(() => { throw reaso
n })
    );
};
```

回调函数决定了接收器（ this ）的稳定状态处理的方式：

- 如果回调函数抛出异常或者返回一个 rejected 状态的 Promise ，则会变成拒绝值（ rejection value ）。
- 否则，接收器的稳定状态（ fulfilled 或者 rejected ）变成了 `finally()` 返回的 Promise 的稳定状态。在某种程度上，我们把 `finally` 从方法链上脱离出来了。

示例**1**（作者 Jake Archibald ）使用 `finally()` 隐藏一个 spinner 。简单的版本：

```
showSpinner();
fetchGalleryData()
.then(data => updateGallery(data))
.catch(showNoDataError)
.finally(hideSpinner);
```

示例**2**（作者 Kris Kowal ）使用 `finally()` 拆分一个测试：

```
let HTTP = require("q-io/http");
let server = HTTP.Server(app);
return server.listen(0)
.then(function () {
    // run test
})
.finally(server.stop);
```

24.15 兼容 ES6 的 Promise 库

现在已经有很多的 Promise 库了。下面的库都遵循 ECMAScript 6 API 标准，这意味着你现在就可以使用 Promise 了，并且以后能轻松迁移到原生的 ES6 上去。

- Stefan Penner 的“ RSVP.js ” 是 ES6 Promise API 的一个超集。
  - Jake Archibald 的“ ES6-Promises ”从 RSVP.js 中提取出了仅包含 ES6 API 的部分。
- Kyle Simpson 的“ Native Promise Only (NPO) ”是“原生 ES6 Promise 的一个 pollyfill ，尽量严格地遵循（没有扩展）规范”。
- Calvin Metcalf 的“ Lie ”是“一个小的高性能的 Promise 库，实现了 Promises/A+ 规范”。
- Kris Kowal 的“ Q.Promise ”实现了 ES6 API 。
- 最后， Paul Millr 的“ ES6 Shim ”包含了 Promise 。

24.16 与传统异步代码对接

当你使用一个 Promise 库的时候，有些时候需要使用非基于 Promise 的异步代码。本节讲解了对于 Node.js 风格的异步函数和 jQuery deferreds ，如何对接。

24.16.1 与 Node.js 对接

Promsie 库 Q 有几个工具方法用于把使用 Node.js 风格 `(err,result)` 回调函数作为参数的函数转换成返回 Promise 的函数（甚至有做反向转换的函数 - 将基于 Promise 的函数转换成接收回调函数的函数）。例如：

```
let readFile = Q.denodeify(FS.readFile);

readFile('foo.txt', 'utf-8')
.then(function (text) {
    ...
});
```

denodify 是一个小型的库，仅提供转换功能，转换成符合 ECMAScript 6 Promise API 标准的形式。

24.16.2 与 jQuery 对接

jQuery 有 deferreds ，和 Promise 很像，但是有几处不同导致无法兼容。deferreds 的方法 `then()` 非常像 ES6 Promsie 的 `then()` （主要不同点：deferreds 并不会在响应器中捕获异常）。所以，可以使用 `Promise.resolve()` 把 jQuery deferred 转换成 ES6 Promise ：

```
Promise.resolve(
    jQuery.ajax({
        url: 'somefile.html',
        type: 'GET'
    }))
.then(function (data) {
    console.log(data);
})
.catch(function (reason) {
    console.error(reason);
});
```

24.17 深入阅读

[1] “ Promises/A+ ”，编辑者： Brian Cavalier 和 Domenic Denicola （ JavaScript Promise 的实际标准）

[2] “ JavaScript Promises: There and back again ”，作者： Jake Archibald （很好的 Promise 介绍文）

[3] “ Promise Anti-Patterns ”，作者： Tao of Code （技巧和技术）

[4] “ Promise Patterns ”，作者： Forbes Lindesay

[5] “ The Revealing Constructor Pattern ”，作者： Domenic Denicola （ Promise 构造器使用了此种模式）

这是一个略微的简化。实际上，元素通过 C.resolve() 转换， C 又由种模式决定。